T H R E E

Declarative Constraints

This is the first of several chapters that cover enforcing business rules with constraints. Declarative constraints provide a core and traditional strategy for business rule enforcement. This chapter explains the how and why of declarative constraints. Chapter 4 illustrates the use of declarative constraints and includes a sample student's data model. Chapter 5 demonstrates how to extract declarative constraint definitions from the database data dictionary. Chapter 6 begins a series of chapters on triggers that enforce constraints procedurally.

This chapter makes frequent reference to a STUDENTS table. The STUDENTS table is part of the sample data model described in Chapter 4. The model description includes an entity relationship diagram, DDL, sample data, plus a description of the business rules enforced.

Oracle supports the following SQL constraints:

- Primary Key constraint
- Unique constraint
- Foreign Key constraint
- Check constraint
- Not Null constraint, which is really a special case of a Check constraint

3.1 Primary Key

The PRIMARY KEY of a table constrains a single column, or set of columns, to a unique and mandatory value—mandatory, meaning that no column in the primary key can ever be null. A table need not have a primary key, but this is the exception; most tables are created with a primary key.

Consider a table that stores information about students. We must be able to identify a student and store data associated with that individual student. Each student must have a row and only one row in the STUDENTS table. Also, each row in the STUDENTS table should identify one, and only one, student. The primary key mechanism enables an application, for example, to properly process student tuition bills. Every student will get one, and only one, tuition bill.

In addition to business rule enforcement, there are other database motivations. Primary keys are an integral part of parent-child relationships that enforce referential integrity. Additionally, a primary key requires an index, a physical structure consuming disk space that, when used properly, provides fast access to the data.

The DDL for the primary key constraint can be embedded within the CREATE TABLE statement. Embedded DDL has two syntax versions: a column constraint clause and a table constraint clause. The primary key constraint can be created separate from the table creation statement. This is accomplished with an ALTER TABLE statement. Creation of the constraint with ALTER TABLE can be done immediately after the table is created. It can also be done after the table is populated.

The Oracle database engine enforces the rule of the primary key constraint. Once you create a table with a primary key, you are saying that: first, all values for the primary key are unique; and second, they must have a value, which means that no column in the primary key can ever be NULL.

Enforcement of a constraint can be temporarily disabled and later enabled. This is accomplished with an ALTER TABLE statement and the constraint options: DISABLE and ENABLE. One reason to load data with a disabled constraint is to reduce the load time. The load time with a disabled constraint will be less because indexes are disabled and therefore not updated.

Within a transaction, an application can temporarily suspend constraint enforcement. In this case a program begins a transaction by setting the constraint to a deferred state. The data is loaded followed by a commit. Upon commit, the constraint is applied by Oracle. This option requires that the constraint be created with the DEFERRABLE keyword.

You can load data into a table with a constraint disabled and, after the load, enable the constraint so that the rule is applied only to new data. The old data, should it violate the business rule, can remain in the table. This strategy to business rule enforcement can apply to data warehouses that must have historical data available for analysis. This option requires that the constraint be enabled with a NOVALIDATE keyword.

Several types of primary key enforcement, such as DEFERRABLE and NOVALIDATE, will affect the type of index used with the primary key constraint. These options will use a nonunique index. A conventional primary key constraint, without ever attempting to violate it, even temporarily within a transaction, will use a unique index.

The fact that a table has a primary key is stored in the data dictionary. We have looked at the data dictionary view, USER_TAB_COLUMNS (see Chapter 1, Section 1.6.2, "Data Dictionary"), which is the dictionary view for looking at column names within a table. We also have views for constraints, these are USER_CONSTRAINTS and USER_CONS_COLUMNS. These views show what tables have constraints, as well as constraint name, type, and status.

Two methods by which you can challenge a primary key constraint is to INSERT a duplicate row or UPDATE a column value forcing a duplicate—in either case, the INSERT or UPDATE statement will fail with an Oracle error.

Declaring constraints is only part of an application. Application code should be integrated with constraint enforcement and handle errors that propagate from constraint violations. If your application is an OLTP system and data entry can cause a duplicate insert, the code should contain graceful error handling. It should produce a meaningful end-user response more meaningful than the generic text of a constraint violation or an alarming stack trace.

3.1.1 Creating the Constraint

This section covers creating the constraint with the column constraint clause, table constraint clause, and the ALTER TABLE statement. This section will use a sample table, TEMP, that has the following description:

```
Name                           Null?    Type
------------------------------ -------- ------------
ID                                      VARCHAR2(5)
NO                                      NUMBER
```

There are several syntax methods and styles for creating a primary key:

1. Column Constraint Clause
2. Table Constraint Clause
3. ALTER TABLE statement

The following discussion addresses the three styles with respect to creating a PRIMARY KEY constraint. Other types of constraints, UNIQUE, FOREIGN KEY, and CHECK, can also be created with each style.

3.1.1.1 COLUMN CONSTRAINT CLAUSE

The following creates a table, TEMP, with two columns. The column ID is the primary key. This is an example of a column constraint clause.

```
SQL> CREATE TABLE temp(id VARCHAR2(5) PRIMARY KEY, no
NUMBER);

Table created.
```

Once the table is created rows are inserted; however, an INSERT with a duplicate value for the column ID will fail. The third INSERT statement is a duplicate and consequently fails.

```
SQL> insert into temp values ('AAA', 1);      First row.

1 row created.

SQL> insert into temp values ('BBB', 2);      Second row.

1 row created.

SQL> insert into temp values ('AAA', 3);      Duplicate.
insert into temp values ('AAA')
*
ERROR at line 1:
ORA-00001: unique constraint (SCOTT.SYS_C006083) violated
```

For the last insert, SQL*Plus flushed an error message to the display. This is the behavior of SQL*Plus. If we are writing code in Java or PL/SQL and anticipate a duplicate insert, we can write error handling code, capture the error condition, and handle it gracefully.

The duplicate insert error message prefix is "ORA". The error number is "-00001." That is not a dash between "ORA" and the number, but "ORA" and a minus one.

The aforementioned error message references "SCOTT.SYS_C006083"; this is the name of the primary key constraint. This name was internally generated by Oracle during execution of the CREATE TABLE TEMP statement. To name the primary key constraint with a column constraint clause:

```
CREATE TABLE temp
(id VARCHAR2(5) CONSTRAINT PRIMARY KEY my_constraint_name,
 no NUMBER);
```

Column constraint clauses are quick and appropriate for ad hoc SQL. Two restrictions on column constraint clauses for primary keys are: (a) no concatenated primary key can be declared and (b) you cannot stipulate the tablespace of the index created on behalf of the constraint. Both concatenated keys and tablespace clauses for indexes are covered later in this chapter.

3.1.1.2 TABLE CONSTRAINT CLAUSE

The table constraint clause is attached at the end of the table definition. Table constraint clauses are part of the CREATE TABLE statement—they just come after all the columns are defined. If there is a syntax error in the constraint clause, the statement fails and no table is created.

The following illustrates, in template form, a CREATE TABLE statement that declares a primary key. The table constraint clause allows multiple constraints to be included with a comma separating each constraint definition.

```
CREATE TABLE temp
(id  VARCHAR2(5),
 no  NUMBER,
CONSTRAINT PRIMARY KEY (id),
CONSTRAINT. . next constraint,
CONSTRAINT. . next constraint) TABLESPACE etc;
```

The following creates the TEMP table, using a table constraint clause.

```
CREATE TABLE temp
(id  VARCHAR2(5),
 no  NUMBER,
CONSTRAINT PRIMARY KEY (id)) TABLESPACE student_data;
```

3.1.1.3 ALTER TABLE STATEMENT

The ALTER TABLE statement is another option for managing constraints. Once you create a table, you can use the ALTER TABLE statement to manage constraints, add columns, and change storage parameters. The ALTER TABLE command regarding constraints is used to perform the following:

Function to Perform	ALTER Syntax
Add a constraint	`ALTER TABLE table_name ADD CONSTRAINT etc`
Drop a constraint	`ALTER TABLE table_name DROP CONSTRAINT etc`
Disable a constraint	`ALTER TABLE table_name DISABLE CONSTRAINT etc`
Enable a constraint	`ALTER TABLE table_name ENABLE CONSTRAINT etc`

The following DDL consists of two DDL statements: a CREATE TABLE statement and an ALTER TABLE statement for the primary key. In this example, the constraint is named PK_TEMP.

```
CREATE TABLE temp (id  VARCHAR2(5), no  NUMBER);

ALTER TABLE temp ADD CONSTRAINT pk_temp PRIMARY KEY (id);
```

The ALTER TABLE command has many options. An approach to remembering the syntax is to consider the information Oracle needs to perform this operation:

- You have to say what table you are altering; you begin with:

  `ALTER TABLE table_name`

- Then, what are you doing? Adding a constraint:

```
ALTER TABLE table_name ADD CONSTRAINT
```

- It is highly recommended but not required that you include a name for the constraint. The constraint name is not embedded in quotes, but will be stored in the data dictionary in upper case. For the table TEMP, append the constraint name PK_TEMP.

```
ALTER TABLE temp ADD CONSTRAINT pk_temp
```

- Denote the type of constraint that will be a PRIMARY KEY, UNIQUE, FOREIGN KEY, or CHECK constraint.

```
ALTER TABLE temp ADD CONSTRAINT pk_temp PRIMARY KEY
```

- There are a few specific options that follow the reference to the constraint type. For a PRIMARY KEY and UNIQUE constraint, designate the columns of the constraint. For a CHECK constraint, designate the rule. The primary key for the TEMP table is created with the following syntax.

```
ALTER TABLE temp ADD CONSTRAINT pk_temp PRIMARY KEY
(ID);
```

- For PRIMARY KEY and UNIQUE constraints we should designate the tablespace name of the index that is generated as a by-product of the constraint—this topic is covered in detail in Section 3.1.3, "The Primary Key Index." To designate the index tablespace, use keywords USING INDEX TABLESPACE. The final syntax is:

```
ALTER TABLE temp ADD CONSTRAINT pk_temp PRIMARY KEY
(ID) USING INDEX TABLESPACE student_index;
```

3.1.2 Naming the Constraint

This section will use a sample table, TEMP, that has the following description:

```
Name                           Null?    Type
------------------------------ -------- ------------
ID                                      VARCHAR2(5)
NO                                      NUMBER
```

The following example creates a table with an unnamed primary key constraint.

```
CREATE TABLE temp(id VARCHAR2(5) PRIMARY KEY, no NUMBER);
```

All constraints have a name and when no name is supplied, Oracle internally creates one with a syntax, SYS_C, followed by a number. The constraint name is important. Troubleshooting frequently requires that we query the data dictionary for the constraint type, table, and column name. For the aforementioned CREATE TABLE statement, a duplicate insert will produce an error.

```
ORA-00001: unique constraint (SCOTT.SYS_C006083) violated
```

The following creates the table and assigns the constraint name PK_TEMP:

```
CREATE TABLE temp
(id  VARCHAR2(5) CONSTRAINT pk_temp PRIMARY KEY,
 no   NUMBER);
```

A duplicate insert here includes the constraint name as well. This constraint name, PK_TEMP, is more revealing as to the nature of the problem.

```
ORA-00001: unique constraint (SCOTT.PK_TEMP) violated
```

Regardless of the syntax form, you can always name a constraint. The following illustrates the TEMP table and primary key constraint with a column constraint clause, table constraint clause, and ALTER TABLE statement, respectively.

```
(1) CREATE TABLE temp
    (id  VARCHAR2(5) CONSTRAINT pk_temp PRIMARY KEY,
      no  NUMBER);

(2) CREATE TABLE temp
    (id  VARCHAR2(5),
     no  NUMBER,
    CONSTRAINT pk_temp PRIMARY KEY (id));

(3) CREATE TABLE temp
    (id  VARCHAR2(5),
     no  NUMBER);
    ALTER TABLE temp ADD CONSTRAINT pk_temp PRIMARY KEY
    (id);
```

Having consistency to constraint names, or following a format, is just as important as naming the constraint. For all stages of a database, be it development, test, or production, it makes sense to have primary key constraints named. Primary key constraints are commonly named with one of two formats. Both formats indicate the constraint type with the PK designation and the table name.

1. PK_*table_name*
2. *table_name*_PK

Suppose you just installed a new application and your end user calls you. Would you rather have the user say, "I ran this application and got an Oracle error that says: SYS_C006083 constraint violation." Or, would you rather hear, "I ran this application and got an Oracle error: PK_TEMP."

If the constraint name is PK_TEMP, you can quickly comfort the caller because you know exactly what the problem is: a primary key constraint violation on that table. Without a meaningful constraint name, you would begin with querying the data dictionary views DBA_CONSTRAINTS to track the table upon which that constraint, SYS_C006083, is declared.

A common oversight when using a data modeling tool is to not name constraints. A data modeler is usually far more concerned with system requirements than constraint names. Having to go back and type in numerous constraint names can be tedious. If you use such a tool, it is worthwhile to do a small demo to see what DDL the tool generates. You want to make sure the final DDL meets your criteria and that you are using the features of the tool that generate the desired DDL. Oracle Designer, very conveniently, will automatically name primary key constraints with a "PK" prefix followed by the table name.

3.1.3 The Primary Key Index

This section covers the index that must always exist as part of the primary key. The topic of tablespace is included. Refer to Chapter 1, Section 1.6.1, "Application Tablespaces," for additional information on the definition, use, and creation of an Oracle tablespace.

This section will use a sample table, STUDENTS, that has the following description:

```
Name                           Null?    Type
------------------------------ -------- ------------
STUDENT_ID                              VARCHAR2(10)
STUDENT_NAME                            VARCHAR2(30)
COLLEGE_MAJOR                           VARCHAT2(15)
STATUS                                  VARCHAR2(20)
```

Whenever you create a primary key, Oracle creates an index on the column(s) that make up the primary key. If an index already exists on those columns, then Oracle will use that index.

Indexes are an integral part of the primary key. Depending on the primary key options, the index may be unique or nonunique. Deferrable primary key constraints use nonunique indexes. Indexes are not used to enforce the business rule of the primary key, but an index is still required. The benefits of

the index are seen with queries against the table. If the primary key constraint is disabled, the index is dropped and query performance suffers.

Tables occupy physical storage. Indexes also use physical storage. The creation of the primary key should designate a tablespace for that index. Because of I/O contention and the fact that indexes grow differently than tables, we always place indexes in separate tablespaces.

The following ALTER TABLE statement creates the primary key, plus designates the tablespace for the index—USING INDEX TABLESPACE is a keyword phrase to this syntax.

```
CREATE TABLE students
 (student_id    VARCHAR2(10),
  student_name  VARCHAR2(30),
  college_major VARCHAR2(15),
  status        VARCHAR2(20)) TABLESPACE student_data;

ALTER TABLE students
  ADD CONSTRAINT pk_students PRIMARY KEY (student_id)
  USING INDEX TABLESPACE student_index;
```

If you do not designate a tablespace for the primary key index, that index is built in your default tablespace. All Oracle user accounts are created with a default tablespace. Tables and indexes created by a user with no tablespace designation fall into this default tablespace.

For example, the following DDL creates an index object in the default tablespace. Because there is no tablespace clause on the CREATE TABLE statement, the table is also created in the default tablespace.

```
CREATE TABLE temp(id VARCHAR2(5) PRIMARY KEY, no NUMBER);
```

The following puts the TEMP table in the STUDENT_DATA tablespace and the primary key in the STUDENT_INDEX tablespace.

```
CREATE TABLE temp(id VARCHAR2(5), no NUMBER)
tablespace STUDENT_DATA;

ALTER TABLE temp ADD CONSTRAINT pk_temp PRIMARY KEY (ID)
USING INDEX TABLESPACE student_index;
```

To create tables and indexes in tablespaces other than your default requires privileges. If you have the RESOURCE role, then you may still have the privilege UNLIMITED TABLESPACE—this privilege is automatically inherited with the RESOURCE role and gives you unlimited tablespace quotas. With UNLIMITED TABLESPACE you are able to create objects in any tablespace including the SYSTEM tablespace. For this reason, the privilege is often revoked from application developers and tablespace quotas are added.

The change made to developer accounts is similar to the change made to the SCOTT account as follows.

```
REVOKE UNLIMITED TABLESPACE FROM SCOTT;
ALTER USER SCOTT QUOTA UNLIMITED ON STUDENT_DATA;
ALTER USER SCOTT QUOTA UNLIMITED ON STUDENT_INDEX;
ALTER USER SCOTT DEFAULT TABLESPACE STUDENT_DATA;
```

Check your account privileges and tablespace quotas with the following SQL. Privileges and roles that are granted to your Oracle account are queried from the data dictionary views USER_ROLE_PRIVS and USER_SYS_PRIVS:

```
column role_priv format a30
SELECT  'ROLE: '||granted_role role_priv
FROM     user_role_privs
UNION
SELECT  'PRIVILEGE: '||privilege role_priv
FROM     user_sys_privs;

ROLE_PRIV
------------------------------
PRIVILEGE: SELECT ANY TABLE
PRIVILEGE: CREATE ANY MATERIALIZED VIEW
ROLE: CONNECT
ROLE: RESOURCE
ROLE: SELECT_CATALOG_ROLE
```

To see tablespace quotas, query USER_TS_QUOTAS:

```
SELECT tablespace_name, max_bytes FROM user_ts_quotas;
```

The aforementioned SQL will return a minus 1 for any tablespace for which you have an unlimited quota. For example:

```
TABLESPACE_NAME                 MAX_BYTES
------------------------------ ----------
STUDENT_DATA                           -1
STUDENT_INDEX                          -1
```

An index is an object that is created in a tablespace. It is a physical structure that consumes disk space. When you create a Primary Key or Unique constraint, an index is either automatically created or an existing index may be reused.

An index is based on a tree structure. Indexes are used by Oracle to execute SELECT statements. The execution of a SELECT using an index is generally faster than a SELECT that does not use an index.

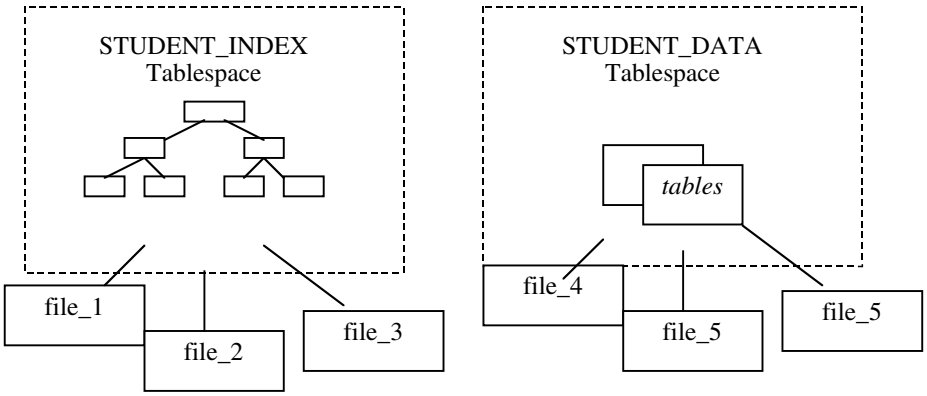

Figure 3–1 *Primary Key Indexes.*

Indexes are generally created in tablespaces separate from tablespaces for tables. Indexes from primary keys make up a portion of all indexes in an application. A tablespace for indexes will have indexes from primary key constraints, indexes from unique constraints, and indexes created to speed up selected queries.

Figure 3–1 illustrates the physical separation of the index structures and table structures. The left side of Figure 3–1 represents a tablespace for indexes. All index structures physically exist in the files for this tablespace. The right side illustrates the allocation of tables. Tables physically exist in the files for the STUDENT_DATA tablespace.

To determine the physical space allocated by the STUDENTS table and the primary key index, query DBA_EXTENTS and DBA_DATA_FILES. The following illustrates a SQL*Plus session query against these views. Because these views begin with DBA, you need either the DBA role or SELECT_CATALOG_ROLE role (refer to Chapter 5, Section 5.1, "What You Can See," for additional information on data dictionary views).

```
SQ1>  SELECT a.extent_id,
  2*         a.segment_name,
  3*         b.file_name,
  4*         round(a.bytes/1024) KBytes
  5*    FROM dba_extents a,
  6*         dba_data_files b
  7*   WHERE segment_name in ('STUDENTS','PK_STUDENTS')
  8*     AND a.file_id=b.file_id;
SQL>
```

The result from the previous code will produce something like the following.

```
EXTENT_ID  SEGMENT_NAME  FILE_NAME                    KBYTES
---------  ------------  --------------------------- ------
        0  STUDENTS      D:\. .\STUDENT_DATA01.DBF       60
        0  PK_STUDENTS   D:\. .\STUDENT_INDEX01.DBF      60
```

The previous output shows that the index created from the primary key initially consumes 60K of disk space in the file STUDENT_INDEX01.DBF. This allocated space is in a file, separate from where the initial 60K of disk space is allocated for the STUDENTS table.

The SEGMENT_NAME column, PK_STUDENTS, is the name of the index, not the constraint. When you create a primary key constraint, an index is created with the same name. That is the situation discussed earlier. The index name can differ if the index is created first and then the constraint is created. When a primary key or unique constraint is created, Oracle looks for an index on the table and on the same columns of the constraint. If such an index exists, that index is used, regardless of the index name.

The following paragraphs explain the aforementioned SQL*Plus query that selects data dictionary information from two views: DBA_EXTENTS and DBA_DATA_FILES.

Objects that require disk space are also called segments. A table is a relational object, but it is also a physical segment—it uses disk space. You query DBA_SEGMENTS to get the physical attributes of tables, partitions, indexes, clusters, and materialized views—any object that requires disk capacity. Stored procedures and sequences are not segments—they are objects that are defined in the data dictionary and their definitions exist in the system tablespace. For every segment, there are one or more extents. An extent is a contiguous set of database blocks.

A table is a segment. The table segment can consume 15 contiguous blocks on disk, where each block is 4K. Blocks 1 through 15, times the 4K block size, yields 60K. This is one extent. There can be other extents. Each extent is a contiguous series of 15 blocks, each block being 4K. Block sizes can vary, and so can the size and number of extents.

The query result shows that there is 60K of contiguous blocks allocated for the index and 60K of contiguous blocks allocated for the STUDENTS table. This is what is first allocated just as a result of creating the table and index. When the 60K fills up, the table or index will extend to another 60K. The datafile for the STUDENT_DATA tablespace is 5M. We are only using 60K of that.

The data dictionary views DBA_SEGMENTS, DBA_EXTENTS, and DBA_DATA_FILES are three of many dictionary-wide views that provide detailed information about space allocation of segments.

The following summarizes the relationship between the primary key constraint and the index.

- An index can be created on any column or set of columns other than the primary key. When we execute the DDL statement that creates a primary key, regardless of the syntax, an index is always created, provided an index on those exact columns does not already exist. A primary key and unique constraint are not the only means of establishing an index. Frequently, many other indexes are created as a means to achieve optimal performance.
- The fact that there is a primary key constraint is wholly defined within the data dictionary. No space is allocated anywhere except the data dictionary tablespace that records this constraint definition. However, the index is an integral part of the constraint. It is an object, takes space, and can be viewed from the data dictionary views USER_OBJECTS and USER_INDEXES, USER_SEGMENTS, and USER_EXTENTS.
- A primary key constraint can be challenged with an INSERT and UPDATE statement—these are the only means by which we can possibly attempt to violate the constraint. The index generated by the primary key constraint does provide a valuable mechanism for optimizing SELECT statements.

3.1.4 Sequences

This section will use a sample table, STUDENTS, that has the following description:

```
Name                           Null?    Type
------------------------------ -------- ------------
STUDENT_ID                              VARCHAR2(10)
STUDENT_NAME                            VARCHAR2(30)
COLLEGE_MAJOR                           VARCHAT2(15)
STATUS                                  VARCHAR2(20)
```

The STUDENTS table with primary key constraint is created with the following DDL.

```
CREATE TABLE students
 (student_id    VARCHAR2(10),
  student_name  VARCHAR2(30),
  college_major VARCHAR2(15),
  status        VARCHAR2(20)) TABLESPACE student_data;

ALTER TABLE students
  ADD CONSTRAINT pk_students PRIMARY KEY (student_id)
  USING INDEX TABLESPACE student_index;
```

For our student system database, what do we use for a STUDENT_ID? One option is to generate a unique student ID for each student that enters the college. We could use social security numbers, information from student visas, even driver's license data. An alternative is for the application code to auto-generate student ID numbers as each student is added to the database.

Databases handle auto-generated columns in a variety of ways. SQL Server uses an IDENTITY column; Oracle uses sequences. We start with creating a sequence that is an object. The existence of that object is stored in the data dictionary. The sequence always has state information, such as current value, and that context remains within the data dictionary. Once you create a sequence you can query the attributes of that sequence from the data dictionary view USER_SEQUENCES.

A sequence has two attributes, NEXTVAL and CURRVAL:

Sequence Attribute	Description
`sequence_name.NEXTVAL`	This evaluates to the next highest value.
`sequence_name.CURRVAL`	This evaluates to the value that was returned from the most recent NEXTVAL call.

We can experiment with sequences right within SQL*Plus as illustrated with the following script. The SQL*Plus session text that follows creates a sequence named MY_SEQUENCE and then uses that sequence to insert values into a TEMP table, also created here. The first statement creates a sequence with defaults, which means that the first time we use that sequence, the value for NEXTVAL will be the number 1.

```
SQL> CREATE SEQUENCE my_sequence;
Sequence created.

SQL> CREATE TABLE temp(n NUMBER);
Table created.

SQL> INSERT INTO temp VALUES (my_sequence.NEXTVAL);
1 row created.

SQL> /                                  Repeat the last insert.
1 row created.

SQL> /                                  Repeat the last insert.
1 row created.

SQL> INSERT INTO temp VALUES (my_sequence.CURRVAL);
1 row created.

SQL> SELECT * FROM temp;        Now, what is in this table?
```

```
         N
----------
         1
         2
         3
         3
```

The keyword START WITH designates the starting point of a sequence. To recreate the sequence with a starting point of 10:

```
SQL> DROP SEQUENCE my_sequence;
Sequence dropped.

SQL> CREATE SEQUENCE my_sequence START WITH 10;
Sequence created.

SQL> SELECT my_sequence.NEXTVAL from dual;

   NEXTVAL
----------
        10
```

If you shut the database down, start it back up, and repeat the previous three INSERT statements, the sequence numbers will continue with 4, 5, and 6. They will continue sequencing because the state of the sequence number is maintained in the data dictionary. When the database is shut down, the state of that sequence number is somewhere in the system tablespace datafile.

Sequences can be created so that the numbers either cycle or stop sequencing at some maximum value. Keywords for this are CYCLE and NOCYCLE. The INCREMENT BY interval can create the sequence to increment by a multiple of any number; the default is 1. The CACHE option pre-allocates a cache of numbers in memory for improved performance. The following illustrates a sequence that will cycle through the values: 0, 5, 10, 15, and 20, then back around to 0.

```
SQL> CREATE SEQUENCE sample_sequence
  2* MINVALUE   0
  3* START WITH 0
  4* MAXVALUE  20
  5* INCREMENT BY 5
  6* NOCACHE
  7* CYCLE;

Sequence created.
```

Sequence numbers are not tied to any table. There is no dependency between any table and a sequence. Oracle has no way to know that you are using a particular sequence to populate the primary key of a specific table.

If you drop and recreate a sequence you may possibly, temporarily, invalidate a stored procedure using that sequence. This would occur if a stored procedure includes the sequence with NEXTVAL or CURRVAL in a PL/SQL or SQL statement. Once the sequence is dropped, the stored procedure becomes invalid. Once the sequence is recreated the procedure will be compiled and validated at run time.

Once you create a sequence you can use it to populate any column in any table. For this reason it is very desirable to carefully name sequences to reflect their use and restrict sequences to the purpose of populating a specific column.

If the primary key is generated by a sequence, the sequence should be named with the following format:

```
table_name_PK_SEQ
```

Using this syntax, a sequence dedicated to the generation of primary key values in the STUDENTS table would be named:

```
STUDENTS_PK_SEQ
```

The view, USER_SEQUENCES, shows the attributes of all sequences in a schema. If we see a sequence named STUDENTS_PK_SEQ, we are quite certain this sequence is used to populate the primary key column of the STUDENTS table. We are certain only because of the sequence name, not from any other data dictionary information.

The complete DDL to create the STUDENTS table, primary key constraint with tablespace clause, and the sequence is the following:

```
CREATE TABLE students
 (student_id    VARCHAR2(10),
  student_name  VARCHAR2(30),
  college_major VARCHAR2(15),
  status        VARCHAR2(20)) TABLESPACE student_data;

ALTER TABLE students
  ADD CONSTRAINT pk_students PRIMARY KEY (student_id)
  USING INDEX TABLESPACE student_index;

CREATE SEQUENCE students_pk_seq;
```

3.1.5 Sequences in Code

You can use the NEXTVAL attribute in any SQL INSERT statement to add a new student. The following Java procedure uses the sequence STUDENTS_PK_SEQ to insert a new student. This code could be called when a student submits, as part of the admission process, an HTML form with the student name, subject major, and status. There is no student ID on the HTML form—the student ID is evaluated as part of the SQL INSERT statement.

```
public void insertStudent(String StudentName,
    String CollegeMajor, String Status)
{
    try
    {
       stmt = conn.prepareStatement
       ("INSERT INTO students (student_id," +
        " student_name, college_major, status)" +
        " VALUES( students_pk_seq.NEXTVAL, :b1, :b2, :b3) ");

       stmt.setString(1, StudentName);
       stmt.setString(2, CollegeMajor);
       stmt.setString(3, Status);
       stmt.execute();
    }
    catch (SQLException e)
    {
       if (e.getErrorCode() == 1)
       {
           System.out.println("We have a duplicate insert.");
       }
    }
}
```

Section 3.1.1, "Syntax Options," illustrates a SQL*Plus session where a duplicate insert is met with an ORA error number of minus 1 and an Oracle error text message. The aforementioned Java procedure includes a try-catch exception handler for the same type of primary key constraint violation. This try-catch handler uses the getErrorCode() method, which returns the five-digit ORA number—in this case, 1. Such an error may be unlikely, but it would not be impossible. Error handling code is supposed to be used infrequently, but it is not supposed to be missing from the application.

The aforementioned Java method, insertStudent(), inserts STUDENTS_PK_SEQ.NEXTVAL as the new student ID for column STUDENT_ID. This expression "NEXTVAL" always evaluates to an integer. Even though the column for STUDENT_ID is string, VARCHAR2(10), the sequence result is implicitly converted to a string by Oracle.

You can manipulate the sequence and use it to build a unique string that satisfies a desired format. We are using a VARCHAR2(10) data type for STUDENT_ID. Suppose we want a STUDENT_ID to be the letter "A" followed by a string of nine digits. We could declare a nine-digit sequence from 1 to 999,999,999—then, concatenate that string with our prefix letter. Our CREATE SEQUENCE statement would first declare the MINVALUE and MAXVALUE using these limits. INSERT statements would then "zero-left-pad" the sequence and concatenate the prefix. The INSERT statement by itself would be the following.

```
INSERT INTO students
VALUES ('A' || LPAD(student_id.NEXTVAL, 9, '0'), etc);
```

Sequence numbers are a safe strategy for guaranteeing unique values. The default CREATE SEQUENCE syntax will generate 10^{27} numbers before you cycle around—this should handle most applications.

The auto-generation feature of sequences should not be a reason to forgo error handling code. An application program using the sequence may be the main vehicle for adding students, but rare events may force inserts from other means, such as SQL*Plus. A rare situation could cause a student to be added "by hand"—possibly due to a problem with a missed student, resolved by an operations person entering the student with SQL*Plus. The operations staff may just use the next highest number in the STUDENTS table for that INSERT. The next time the application runs and enters a new student, it will generate the next sequence value and collide with what operations did with SQL*Plus.

Problems with sequences can also occur when applications are migrated and a sequence is inadvertently dropped and recreated. In this case the sequence starts over with a student ID of 1 and causes a duplicate insert. With regard to error handling, the sequence does provide uniqueness, but mistakes happen. As rare as a duplicate insert might be, a graceful capture of a primary key constraint violation will always save hours of troubleshooting.

3.1.6 Concatenated Primary Key

In this section we make use of a table, STUDENT_VEHICLES, that stores information on vehicles that students keep on campus. The table description is:

```
Name                                      Null?    Type
----------------------------- -------- ------------
STATE                         NOT NULL VARCHAR2(2)
TAG_NO                        NOT NULL VARCHAR2(10)
VEHICLE_DESC                  NOT NULL VARCHAR2(30)
STUDENT_ID                    NOT NULL VARCHAR2(10)
PARKING_STICKER               NOT NULL VARCHAR2(10)
```

Your client, the motor vehicle department of the school, expresses a need, "We want to track student vehicles on campus. Students come from all over the country with cars. Students are allowed to keep cars on campus, but the college needs to track vehicles. The college issues a parking sticker that permits the car to remain on campus."

Data model analysis begins with looking at vehicle tag information. License's tags are issued by each state, so we can be certain that all vehicle license tag numbers issued by New York are unique within that state and all tag numbers from California are unique within that state. Tag numbers are short-string combinations of letters and numbers.

We can assume the following rule: within each state, all tags within that state are unique. Consequently, the combination of state abbreviation and the tag number of any student's vehicle will be always unique among all students. This rule is enforced with a combination of these two columns forming a concatenated primary key.

Below is sample data for California (CA) and New York (NY) license numbers for three students. There is no rule about a student registering more than one vehicle on campus. As you can see, student A104 has two registered vehicles.

```
STATE TAG_NO   VEHICLE_DESC  STUDENT_ID PARKING_STICKER
----- -------- ------------  ---------- ---------------
CA    CD 2348  1977 Mustang  A103       C-101-AB-1
NY    MH 8709  1989 GTI      A104       C-101-AB-2
NY    JR 9837  1981 Civic    A104       C-101-AB-3
```

We will store this information in a new table called STUDENT_VEHICLES. We know there will be other columns of interest (e.g., vehicle registration information); for now, we'll just include vehicle description, the student who uses the car, and the number of the parking sticker issued by the campus police. The key here is that, in the real world, we can definitely say that the combination of a state abbreviation and license tag number is always unique—this makes (STATE, TAG_NO) a concatenated primary key. The DDL for this table is:

```
CREATE TABLE student_vehicles
 (state           VARCHAR2(2),
  tag_no          VARCHAR2(10),
  vehicle_desc    VARCHAR2(20),
  student_id      VARCHAR2(10),
  parking_sticker VARCHAR2(10)) TABLESPACE student_data;

ALTER TABLE student_vehicles
  ADD CONSTRAINT pk_student_vehicles
  PRIMARY KEY (state, tag_no)
  USING INDEX TABLESPACE student_index;
```

Concatenated primary keys and sequence-generated keys do not mix. A single column is all that is needed for a sequence-generated primary key. This table should not incorporate a sequence; it stores information on vehicles that naturally and uniquely distinguishes each vehicle by state and tag number.

During your initial design, you may be told by your client that certain values will always be unique. Your client may say, for example with an inventory system, that the combination of COMPANY and PART_NO will always be unique. You go with this concept and construct a concatenated primary key. Months later, you learn that there are exceptions—sometimes this is as much a revelation to the client as it is to you. Your approach then is to rebuild the tables and add a new column for the sequence.

Some modelers will, as a standard practice, make every primary key a sequence-generated column. This works but, to a degree, discards reality. For example, there is no doubt that a state abbreviation and a tag number, in the real world, are unique and truly can be used to uniquely identify the attributes of any vehicle on campus.

When the de facto standard in a project is to make all primary keys sequence-generated, you have to pay closer attention to how you query the table because you may frequently query the table using real-world attributes like state abbreviation and tag number. These columns might not have indexes, unless you specifically create them. The column with the sequence has an index because it has a primary key; however, other columns may not have, but should have indexes.

3.1.7 Extra Indexes with Pseudo Keys

The STUDENT_VEHICLES table has a natural primary key—it seems natural to assume that the combination of state abbreviation and license number will always be unique. Suppose we add a pseudo key, insisting on a new column called VEHICLE ID. We'll make VEHICLE_ID the primary key. We then have the following table.

```
CREATE TABLE student_vehicles
 (vehicle_id      NUMBER,
  state           VARCHAR2(2),
  tag_no          VARCHAR2(10),
  vehicle_desc    VARCHAR2(20),
  student_id      VARCHAR2(10),
  parking_sticker VARCHAR2(10)) TABLESPACE student_data;

ALTER TABLE student_vehicles
  ADD CONSTRAINT pk_student_vehicles
  PRIMARY KEY (vehicle_id)
  USING INDEX TABLESPACE student_index;

CREATE SEQUENCE student_vehicles_pk_seq;
```

Suppose we have the same sample data as Section 3.1.6, "Concatenated Primary Key." The primary key, VEHICLE_ID, is a sequence number that starts with 1. After adding three vehicles we have the following data.

```
VEH_ID STATE TAG_NO   VEHICLE_DESC  STUDENT_ID PARKING_STICKER
------ ----- -------- ------------  ---------- ---------------
     1 CA    CD 2348  1977 Mustang  A103       C-101-AB-1
     2 NY    MH 8709  1989 GTI      A104       C-101-AB-2
     3 NY    JR 9837  1981 Civic    A104       C-101-AB-3
```

The following SQL queries the table using the primary key column in the query. This query will use the index.

```
SELECT * FROM student_vehicles WHERE vehicle_id = '1';
```

The execution plan, shown here, will include the index.

```
Execution Plan
----------------------------------------------------------
0    SELECT STATEMENT Optimizer=CHOOSE
1  0   TABLE ACCESS (BY INDEX ROWID) OF 'STUDENT_VEHICLES'
2  1     INDEX (UNIQUE SCAN) OF 'PK_STUDENT_VEHICLES'
         (UNIQUE)
```

What if the application code relies on the columns for state and license number? This is a likely possibility. A lookup of information based on a state and license plate number could be dependent on fields selected on an HTML form. The application developer could have used the primary key, but chose to use the other columns. The following SQL will return the same information as the previous select statement.

```
SELECT *
  FROM students
 WHERE state = 'CA'
   AND tag_no = 'CD 2348';
```

The difference is that the first query will likely run faster. If there are a significant number of students, the first query will use the index of the primary key. The second query will not use an index because there is no index on columns STATE and TAG_NO. The execution plan shows a table scan that will cause considerable wait time for the end user.

```
Execution Plan
----------------------------------------------------------
0    SELECT STATEMENT Optimizer=CHOOSE
1    0   TABLE ACCESS (FULL) OF 'STUDENT_VEHICLES'
```

When a pseudo key is created, as earlier, and there is a natural primary key in the table, there is the risk that the application developed will take fields, such as STATE and TAG NO from a form (i.e., an HTML form or client GUI form) and use those fields to query the database. When this happens, the performance of that query is slow unless actions are taken to identify these table scans and resolve them with additional indexes.

The decision to add a pseudo key is not being judged here—the issue is to recognize when and if the primary key, whatever that key may be, is being used and when it is not being used. There are three options when the application is not using the sequence primary key.

1. Don't use the pseudo index, VEHICLE_ID. It provides no benefit because there is a natural primary key built into the data: STATE and TAG_NO.
2. Create an index on the columns STATE and TAG_NO.
3. Create a concatenated UNIQUE constraint on columns STATE and TAG_NO; the mere creation of this unique constraint will enforce the natural business rule that these columns are unique, plus create an index on these columns.

Why would someone choose to use a sequence rather than columns that, in the real world, identify a unique row? This is a decision made by the data modeler. Maybe the natural primary key is many columns—financial applications that deal with financial instruments and trading of those instruments have so many attributes that the natural primary key of a table is many columns. That doesn't mean those columns can't be a primary key. An excessive number of columns in a primary key means large indexes and possible foreign keys that, if we have indexes on those, means larger indexes. Many columns in the primary key may not be desirable.

Secondly, maybe the modeler has doubts as to whether, in the future, those columns will remain unique. Sometimes end users say the data is unique; the data modeler creates a primary key only to find out later that the real data does, in fact, contain duplicates. So, reverting to a concatenated primary key may not be the first choice.

Suppose you stick with the pseudo key, such as VEHICLE_ID. Given this approach, we can still consider the business rule that, within the STUDENT_VEHICLES table, the STATE and TAG_NO are still unique and you can construct a UNIQUE constraint on those columns. The UNIQUE constraint does generate an index and consequently the query following will have the potential to perform at optimal levels with the use of the index generated through the unique constraint.

```
SELECT *
  FROM student_vehicles
 WHERE state = 'CA'
   AND tag_no = 'CD 2348';
```

The DDL for the scenario to implement the pseudo sequence plus a concatenated unique constraint on STATE and TAG_NO is the following.

```
CREATE TABLE student_vehicles
 (vehicle_id      NUMBER,
  state           VARCHAR2(2),
  tag_no          VARCHAR2(10),
  vehicle_desc    VARCHAR2(20),
  student_id      VARCHAR2(10),
  parking_sticker VARCHAR2(10)) TABLESPACE student_data;

ALTER TABLE student_vehicles
  ADD CONSTRAINT pk_student_vehicles
  PRIMARY KEY (vehicle_id)
  USING INDEX TABLESPACE student_index;

CREATE SEQUENCE student_vehicles_pk_seq;

ALTER TABLE student_vehicles
  ADD CONSTRAINT uk_student_vehicles_state_tag
  UNIQUE (state, tag_no)
  USING INDEX TABLESPACE student_index;
```

If you do not expect state and license number to be or remain unique, you can always create a nonunique index on these columns. So, rather than add a unique constraint as earlier, replace that with the following, which only creates an index.

```
CREATE INDEX student_vehicles_state_tag
  ON student_license(state, tag_no)
  TABLESPACE student_index;
```

In summary, when your approach to primary keys is, by default, to always use sequence-generated primary keys, consider how you query the table. Are there attributes within the table to which a unique constraint can be applied? If so, create a unique constraint. For all other columns, look at how the table is accessed and add additional indexes to optimize performance.

3.1.8 Enable, Disable, and Drop

You can drop a primary key constraint with the following.

```
ALTER TABLE <table_name> DROP CONSTRAINT
<constraint_name>;
```

This drops the constraint and drops the index associated with that constraint.

Oracle will not allow you to drop a primary key to which there is a referencing foreign key. If a table has a referencing foreign key you will get this error.

```
ORA-02273: this unique/primary key is referenced
by some foreign keys
```

You can DROP CASCADE the constraint. This will drop the primary key constraint and all foreign keys that reference that parent. This does not require that the foreign key constraint be declared with the CASCADE option. You can always drop a primary key with the CASCADE option, but it is permanent. The primary key and foreign key constraints are deleted from the data dictionary. The CASCADE option is:

```
ALTER TABLE state_lookup
DROP CONSTRAINT state_lookup CASCADE;
```

You can check all referencing foreign keys to a primary key constraint with the script MY_CHILDREN_ARE in Chapter 5, Section 5.6.5.

A less drastic measure is to disable a constraint. The CASCADE restriction applies as well to disabling constraints. If there are referencing foreign keys, you can disable a primary key constraint with:

```
ALTER TABLE state_lookup
DISABLE CONSTRAINT state_lookup CASCADE;
```

A disabled constraint is still defined in the data dictionary, it is just not being enforced. Furthermore, the status is set to DISABLED. With the CASCADE option, the status of the primary key and all referencing foreign key constraints are set to a DISABLED status.

The index, after a primary key constraint is disabled, is gone—deleted from the data dictionary; however, it is immediately rebuilt when the constraint is enabled.

```
ALTER TABLE state_lookup ENABLE CONSTRAINT
pk_state_lookup;
```

This ALTER TABLE statement recreates the index and sets the primary key constraint to ENABLED. The foreign key constraints are still disabled. Each of these has to be enabled with:

```
ALTER TABLE students ENABLE CONSTRAINT fk_students_state;
```

An index on a primary key column cannot be dropped—it would be dropped with a DROP or DISABLE constraint. Indexes can be rebuilt with no affect on the constraint with:

```
ALTER INDEX pk_state_lookup REBUILD;
```

The ENABLE and DISABLE keywords can be appended to any constraint declaration. Attaching DISABLE creates the constraint and sets the state to DISABLED. It can be enabled any time. The ENABLE keyword is the default for all constraint clauses.

3.1.9 Deferrable Option

A primary key constraint can be created with the option DEFERRABLE. This option permits an application program to disable the constraint during a load. The assumption is that manipulation of the table will occur, but when the load is complete the data will conform to the rules of the primary key constraint. The application will COMMIT the loaded data. At that COMMIT, the constraint is enforced. Should the program have left invalid data in the table—data that violates the constraint—the transaction is rolled back.

Scenario 1 Bad Data Is Loaded with Deferrable

A stored procedure loads 1,000 rows.There are duplicates.All 1,000 rows are loaded.The program does a commit.The transaction is rolled back because the data violates the constraint.

Scenario 2 Bad Data Is Loaded without Deferrable

A stored procedure loads 1000 rows.There are duplicates in this data.The program proceeds to load the data and at some point a duplicate is inserted.The insert fails.The application can choose to ignore this single insert (using exception handling code) and continue with the remaining data.The program is also capable of rolling back the transaction.

The DEFERRABLE option is created with the following syntax—a sample table PARENT is created with two columns.

```
CREATE TABLE parent
(parent_id   NUMBER(2),
 parent_desc VARCHAR2(10));

ALTER TABLE parent ADD CONSTRAINT pk_parent PRIMARY KEY
(parent_id) DEFERRABLE;
```

The aforementioned constraint definition is the same as this next statement:

```
ALTER TABLE parent ADD CONSTRAINT pk_parent PRIMARY KEY
(parent_id) DEFERRABLE INITIALLY DEFERRED;
```

Scenario 1 is shown with the following code. This PL/SQL block loads duplicate data. After the load, but before the COMMIT, a duplicate is DELETED. This block completes with success because the duplicate was removed prior to the commit.

```
DECLARE
BEGIN
    EXECUTE IMMEDIATE 'SET CONSTRAINTS ALL DEFERRED';

    INSERT INTO parent values (1,'A');
    INSERT INTO parent values (1,'B');
    INSERT INTO parent values (3,'C');
    INSERT INTO parent values (4,'D');

    DELETE FROM parent WHERE parent_desc = 'B';
    COMMIT;
END;
```

If this block had not removed the duplicate, an Oracle error would occur. That error is the same error as a primary key constraint violation. The following block accomplishes the same task, loading the same data. This scenario does not use the deferrable option. Rather, all good data remains in the PARENT table.

```
BEGIN
    BEGIN
        INSERT INTO parent values (1, 'A');
    EXCEPTION WHEN DUP_VAL_ON_INDEX THEN null;
    END;
    BEGIN
        INSERT INTO parent values (1, 'B');
    EXCEPTION WHEN DUP_VAL_ON_INDEX THEN null;
    END;
    BEGIN
        INSERT INTO parent values (3, 'C');
    EXCEPTION WHEN DUP_VAL_ON_INDEX THEN null;
    END;
    BEGIN
        INSERT INTO parent values (4, 'D');
    EXCEPTION WHEN DUP_VAL_ON_INDEX THEN null;
    END;

    COMMIT;
END;
```

The data remaining in the table is the first, third, and fourth row. The DEFERRABLE option loads everything. Then the constraint is applied and the transaction is committed or rolled back. The second scenario provides the option to roll back a duplicate insert. Alternatively, the application can skip the duplicate insert and continue processing good data.

The deferrable option should not be used to permit the table to be a work area. Scenario 1 seems to permit the application to load data, and then manipulate that data, cleaning out bogus records. Hopefully, the data will be good upon a commit. Alternatives exist. Temporary tables can be created for loading data for manipulation. After manipulation and cleansing of the data, the rows are inserted into the production table. Temporary tables can persist for the life of a session or transaction. The following DDL creates a temporary table that can be used as a private in-memory table that persists for the duration of a transaction.

```
CREATE GLOBAL TEMPORARY TABLE
parent_temp
(parent_id   NUMBER(2),
 parent_desc VARCHAR2(10)) ON COMMIT DELETE ROWS;
```

Scenario 1 can now use the temporary table for massaging the raw data. When data is moved into the permanent table, the commit deletes rows from the temporary table. The temporary table is private to this transaction. The following PL/SQL block is Scenario 1 using a temporary table with no deferrable option.

```
BEGIN
    INSERT INTO parent_temp values (1,'A');
    INSERT INTO parent_temp values (1,'B');
    INSERT INTO parent_temp values (3,'C');
    INSERT INTO parent_temp values (4,'D');

    DELETE FROM parent_temp WHERE parent_desc = 'B';

    INSERT INTO parent SELECT * FROM parent_temp;
    COMMIT;
END;
```

The DEFERRABLE option enforces the constraint with each DML statement. This is the default behavior of the constraint without the DEFERRABLE option; however, the DEFERRABLE does provide for the following:

- An application can SET CONSTRAINTS ALL DEFERRED, load data, and have the constraint enforced when the transaction completes with a COMMIT statement.
- The constraint can be disabled with an ALTER TABLE statement to DISABLE the constraint. Data can be loaded. The constraint can be enabled with NOVALIDATE, leaving duplicates in the table but enforcing the constraint for future DML (see Section 3.1.10, "NOVALIDATE.")

The DEFERRABLE option can be declared with the following attribute:

```
DEFERRABLE INITIALLY DEFERRED
```

The INITIALLY DEFERRED feature defaults to enforcing the constraint only when a transaction commits. When this is set, each DML statement is not individually enforced. Enforcement takes place only when the transaction completes. This option provides the following:

- An application begins a transaction knowing that the constraint is enforced upon the commit. The application loads data, then commits. The entire transaction is accepted or rejected. The code contains INSERT statements and a COMMIT. There is nothing in the code to reflect that the INSERTS are validated only upon commit. Prior to a commit, the application can specifically check for validation with the following.

```
EXECUTE IMMEDIATE 'SET CONSTRAINTS ALL IMMEDIATE';
```

The following summarizes DEFERRABLE option.

1. You can declare the constraint DEFERRABLE, which has the following attributes in the data dictionary USER_CONSTRAINTS view.

   DEFERRABLE = DEFERRABLE
   DEFERRED = IMMEDIATE

   This option means that the constraint is enforced with each DML. You can write code, as shown earlier, that directs Oracle to refrain from constraint enforcement until a COMMIT or a ROLLBACK. This requires a SET CONSTRAINTS statement in your code.
2. You can declare the constraint with the option DEFERRABLE INITIALLY DEFERRED. This sets the constraint in the data dictionary to:

   DEFERRABLE = DEFERRABLE
   DEFERRED = DEFERRED

3. This option means that the default behavior for a transaction is to enforce constraints when the transaction does a COMMIT or ROLLBACK. A PL/SQL program would look like any other program with this option. One would have to look into the status of the constraint to see when the constraint is enforced. An application can choose to enforce constraints with the SET CONSTRAINTS ALL IMMEDIATE command. Without this statement, the constraint will be enforced when the transaction completes.
4. You can ALTER the DEFERRED status of a constraint. If you declare it as DEFERRABLE, then it has the status of:

   DEFERRABLE = DEFERRABLE
   DEFERRED = IMMEDIATE

You can execute the following:

```
ALTER TABLE table_name MODIFY CONSTRAINT
constraint_name INITIALLY DEFERRED.
```

This changes the state to:

DEFERRABLE = DEFERRABLE
DEFERRED = DEFERRED

5. You can ALTER the DEFERRED status of a constraint declared INITIALLY DEFERED with the following:

```
ALTER TABLE table_name MODIFY CONSTRAINT
constraint_name INITIALLY IMMEDIATE.
```

This changes the state to:

DEFERRABLE = DEFERRABLE
DEFERRED = IMMEDIATE

As always, you can disable the constraint and then enable it with ALTER TABLE commands.

3.1.10 NOVALIDATE

The NOVALIDATE option allows nonconforming data to be loaded and left in the table while the rule of the constraint is enabled only for future inserts. This option can be used in data warehouse systems where management must have historical data for analysis. Historical data will frequently violate present day business rules.

To load noncompliant data, the constraint must be initially created with the deferrable option. Prior to loading historical data, the constraint must be disabled. The following creates a table with a deferrable primary key constraint. Prior to the load, the constraint is disabled. Afterward, the constraint is enabled with the NOVALIDATE option. From this point forward, the historical data remains but all new inserts will be constrained to the rules of the primary key.

```
CREATE TABLE parent
(parent_id   NUMBER(2),
 parent_desc VARCHAR2(10));

ALTER TABLE parent ADD CONSTRAINT pk_parent PRIMARY KEY
(parent_id) DEFERRABLE;

ALTER TABLE parent DISABLE CONSTRAINT pk_parent;

BEGIN
```

```
    INSERT INTO parent values (1,'A');
    INSERT INTO parent values (1,'B');
    INSERT INTO parent values (3,'C');
    INSERT INTO parent values (4,'D');
END;

ALTER TABLE parent ENABLE NOVALIDATE CONSTRAINT pk_parent;
```

After the aforementioned PL/SQL block executes, duplicates exist in the tables; however, all new inserts must conform to the primary key.

3.1.11 Error Handling in PL/SQL

A duplicate insert in PL/SQL is easily captured with the PL/SQL built-in exception. The exception name is

```
DUP_VAL_ON_INDEX
```

Including an exception handler will allow an application to handle the rare case of a duplicate. The following stored procedure returns a Boolean, indicating a failure if the insert is a duplicate.

```
CREATE OR REPLACE
FUNCTION insert_parent
    (v_id NUMBER, v_desc VARCHAR2) RETURN BOOLEAN
IS
BEGIN
    INSERT INTO parent VALUES (v_id, v_desc);
    return TRUE;
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN return FALSE;
END;
```

3.2 UNIQUE

In this section we use a table that stores student information. The table description is:

```
Name                           Null?    Type
------------------------------ -------- ------------
STUDENT_ID                              VARCHAR2(10)
STUDENT_NAME                            VARCHAR2(30)
COLLEGE_MAJOR                           VARCHAR2(15)
STATUS                                  VARCHAR2(15)
STATE                                   VARCHAR2(2)
LICENSE_NO                              VARCHAR2(30)
```

The UNIQUE constraint is applied to a column, or set of columns, to enforce the following rule: If a value exists, than that value must be unique.

This definition sounds similar to a PRIMARY KEY constraint. The following is a comparison between the primary key and unique constraints.

- A table can have just one primary key constraint but a table can have more than one unique constraint.
- A column that is part of a primary key can never have a NULL value. A column that is part of a unique constraint can be null. If a column has a unique constraint, there can be many rows with a NULL for that column. The values that are not null must be unique.
- When we create a primary key we create or reuse an index. The same is true for UNIQUE constraints.
- The primary key and unique constraint columns can be the parent in a foreign key relationship. The columns of a foreign key constraint frequently refer back to columns of a primary key constraint. They can also refer back to columns of a unique constraint.

The following DDL creates a concatenated unique constraint on columns (A, B):

```
CREATE TABLE temp (pk NUMBER PRIMARY KEY, a NUMBER, b NUMBER);
ALTER TABLE temp ADD CONSTRAINT uk_temp_a_b UNIQUE (a, b);
```

NULLs are permitted in any column provided the data that does exist qualifies as unique. The following INSERT statements are valid.

```
-- UNIQUE Constraint on last 2 columns.
INSERT INTO temp VALUES (1,    1,    1);
INSERT INTO temp VALUES (2,    2,    1);
INSERT INTO temp VALUES (3,    1,    2);
INSERT INTO temp VALUES (4, NULL, NULL);
INSERT INTO temp VALUES (5, NULL, NULL);
INSERT INTO temp VALUES (6,    1, NULL);
INSERT INTO temp VALUES (7, NULL,    1);
INSERT INTO temp VALUES (8,    2, NULL);
```

The following duplicates the last insert and raises a constraint violation.

```
SQL> insert into temp values (9, 2, NULL);
insert into temp values (9, 2, NULL)
*
ORA-00001: unique constraint (SCOTT.UK_TEMP_A_B) violated
```

Notice that the prefix "ORA" and a minus 1 for the error code are identical to the primary key constraint violation.

3.2.1 Combining NOT NULL, CHECK with UNIQUE Constraints

A NOT NULL constraint is sometimes added to the UNIQUE constraint. This additional requirement stipulates that the column values must be unique and that NULLs are not allowed. The following illustrates this case.

```
CREATE TABLE temp (pk NUMBER PRIMARY KEY, a NUMBER NOT
NULL);
ALTER TABLE temp ADD CONSTRAINT uk_temp_a UNIQUE (a);
```

Concatenated unique constraints are often supplemented with a CHECK constraint that prevents the condition: one column is NULL and one column has a value. This is enforced following this example.

```
CREATE TABLE temp (pk NUMBER PRIMARY KEY, a NUMBER, b NUMBER);
ALTER TABLE temp ADD CONSTRAINT uk_temp_a_b UNIQUE (a, b);
ALTER TABLE temp ADD CONSTRAINT ck_temp_a_b CHECK
    ((a IS NULL AND B IS NULL) OR
     (a IS NOT NULL AND b IS NOT NULL));
```

The aforementioned CHECK and UNIQUE constraint combination allows the first two inserts, but rejects the second two inserts.

```
INSERT INTO temp VALUES (6,    1,    1);  -- successful
INSERT INTO temp VALUES (7, NULL, NULL);  -- successful
INSERT INTO temp VALUES (6,    1, NULL);  -- fails
INSERT INTO temp VALUES (7, NULL,    1);  -- fails
```

Combining NOT NULL and CHECK constraints with UNIQUE constraints allows for several options.

- The column values, when data is present, must be unique. This is enforced with the UNIQUE constraint.
- Any column or all columns in the UNIQUE constraint can be mandatory with NOT NULL constraints on individual columns.
- Combinations of NULL and NOT NULL restrictions can be applied using a CHECK constraint and Boolean logic to dictate the rule.

3.2.2 Students Table Example

Consider a table, STUDENTS, that includes columns for student driver's license information. Not every student has a license—this has to be considered. Our rules are:

- Every student must have a unique identification number or, STUDENT_ID—this is the primary key.
- If a student has a driver's license the concatenation of license number and state abbreviation must be unique. The columns for state and license are not mandatory and do not have a NOT NULL constraint.
- The final CHECK constraint ensures that should a student have a license, the state and license values are both entered into the system.

The following DDL creates the STUDENTS table with a concatenated UNIQUE constraint on STATE and LICENSE_NO.

```
CREATE TABLE students
 (student_id    VARCHAR2(10) NOT NULL,
  student_name  VARCHAR2(30) NOT NULL,
  college_major VARCHAR2(15) NOT NULL,
  status        VARCHAR2(15) NOT NULL,
  state         VARCHAR2(2),
  license_no    VARCHAR2(30)) TABLESPACE student_data;

ALTER TABLE students
  ADD CONSTRAINT pk_students PRIMARY KEY (student_id)
  USING INDEX TABLESPACE student_index;

ALTER TABLE students
  ADD CONSTRAINT uk_students_license
  UNIQUE (state, license_no)
  USING INDEX TABLESPACE student_index;

ALTER TABLE students
   ADD CONSTRAINT ck_students_st_lic
   CHECK ((state IS NULL AND license_no IS NULL) OR
          (state IS NOT NULL AND license_no is NOT NULL));
```

The following paragraphs summarize key points about the above DDL.

- We call the primary key constraint PRIMARY KEY. The unique constraint is just "UNIQUE." In conversation we often refer to "unique key" constraints, but when writing DDL, leave off the "KEY" part. Also, unique constraints are often named with a prefix of "UK." The primary key, foreign key, and unique constraint all work together to enforce referential integrity. But the DDL syntax for a unique constraint does not include the "KEY" keyword.
- The unique constraint causes an index to be created; therefore, we have included the tablespace as the location for this index (if an index on these columns has already been created, then the constraint

will use that index). For all the DDL in this text, the tables are created in a STUDENTS_DATA tablespace and all indexes are created in a STUDENTS_INDEX tablespace—a fairly standard practice.

- The unique constraint is named UK_STUDENTS_LICENSE. Constraint names are limited to 30 characters. Constraint names in this text are preceded with a prefix to indicate the constraint type. This is followed by the table name. For a primary key constraint, that is all we need. For other constraints, we try to append the column name—this can be difficult because table names and column names may be up to 30 characters. Sometimes you must abbreviate. Most important, the name of the constraint should clearly indicate the constraint type and table. The column names included in the constraint can be short abbreviations—this approach helps when resolving an application constraint violation.

3.2.3 Deferrable and NOVALIDATE Options

Similar to primary key constraints, a UNIQUE constraint can be declared as deferrable. The constraint can be disabled and enabled with ALTER TABLE statements. All the options described in Section 3.1.9, "Deferrable Option," and 3.1.10, "NOVALIDATE," are applicable to UNIQUE constraints.

3.2.4 Error Handling in PL/SQL

A duplicate insert, for a primary key and unique constraint, is captured with the PL/SQL built-in exception. The exception name is

```
DUP_VAL_ON_INDEX
```

The following procedure inserts a student and captures unique constraint errors. The code for a check constraint error is also captured. The check constraint error number is mapped to a declared exception. The procedure then includes a handler for that exception. The primary key and unique constraint errors have the predefined exception declared in the language. Other types of errors, such as check constraint errors, need exceptions declared that must be mapped to the Oracle error number. The check constraint error number is minus 2290.

```
 CREATE OR REPLACE PROCEDURE
    insert_student(v_student_id VARCHAR2,
        v_student_name VARCHAR2,
        v_college_major VARCHAR2,
        v_status VARCHAR2,
        v_state VARCHAR2,
        v_license_no VARCHAR2)
IS
```

```
    check_constraint_violation exception;
    pragma exception_init(check_constraint_violation, -2290);
BEGIN
    INSERT INTO students VALUES
        (v_student_id, v_student_name,
         v_college_major, v_status,
         v_state, v_license_no);

    dbms_output.put_line('insert complete');
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        dbms_output.put_line('PK or unique const violation');
    WHEN check_constraint_violation THEN
        dbms_output.put_line('check constraint violation');
END;
```

To test the error handling logic, use SQL*Plus to EXECUTE the procedure with some data. The first two inserts fail because the state and license violate the check constraint rule: both are NULL or both are NOT NULL.

The third and fourth inserts work. The last insert fails because it violates the unique constraint, which is a duplicate of the prior insert.

```
insert_student('A900','Ann','Math','Degree','CA',NULL);
check constraint violation

insert_student('A900','Ann','Math','Degree',NULL,'ABC');
check constraint violation

insert_student('A900','Ann','Math','Degree',NULL,NULL);
insert complete

insert_student('A902','Joe','Math','Degree','CA','ABC');
insert complete

insert_student('A903','Ben','Math','Degree','CA','ABC');
PK or unique const violation
```

3.3 Foreign Key

Foreign key constraints enforce referential integrity. A foreign key constraint restricts the domain of a column value. An example is to restrict a STATE abbreviation to a limited set of values in another control structure—that being a parent table.

The term "lookup" is often used when referring to tables that provide this type of reference information. In some applications, these tables are cre-

ated with this keyword—a practice we'll use here with the example STATE_LOOKUP.

Start with creating a lookup table that provides a complete list of state abbreviations. Then use referential integrity to ensure that students have valid state abbreviations. The first table is the state lookup table with STATE as the primary key.

```
CREATE TABLE state_lookup
 (state      VARCHAR2(2),
  state_desc VARCHAR2(30)) TABLESPACE student_data;

ALTER TABLE state_lookup
  ADD CONSTRAINT pk_state_lookup PRIMARY KEY (state)
  USING INDEX TABLESPACE student_index;
```

To insert a few rows:

```
INSERT INTO state_lookup VALUES ('CA', 'California');
INSERT INTO state_lookup VALUES ('NY', 'New York');
INSERT INTO state_lookup VALUES ('NC', 'North Carolina');
```

We enforce referential integrity by implementing the parent–child relationship, graphically shown in Figure 3–2.

Figure 3–2 shows a one-to-many relationship between the STATE_LOOKUP table and the STUDENTS table. The STATE_LOOKUP table defines the "universal set" of state abbreviations—each state being represented once in that table; hence, a primary key on the STATE column of STATE_LOOKUP.

A state from the STATE_LOOKUP table can appear multiple times in the STUDENTS table. There can be many students from a single state. Hence referential integrity implements a one-to-many relationship between STATE_LOOKUP and STUDENTS.

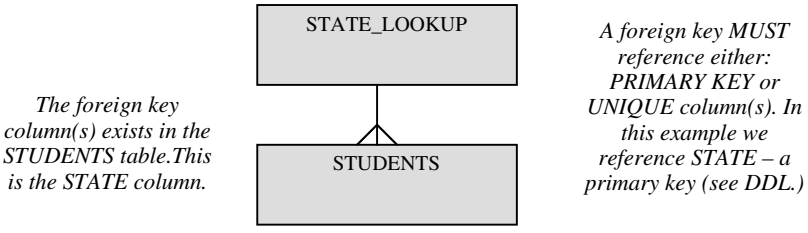

Figure 3–2 *Foreign Key with State Lookup.*

The foreign key also ensures the integrity of the STATE column in the STUDENTS table—a student with a driver's license will always have a state abbreviation that is a member of the STATE_LOOKUP table.

The foreign key constraint is created on the child table. Following is the STUDENTS table with a foreign key constraint. With an ALTER TABLE statement, we declare the column STATE to have a foreign key constraint that references the primary key column of the STATE_LOOKUP table.

```
CREATE TABLE students
 (student_id    VARCHAR2(10) NOT NULL,
  student_name  VARCHAR2(30) NOT NULL,
  college_major VARCHAR2(15) NOT NULL,
  status        VARCHAR2(20) NOT NULL,
  state         VARCHAR2(2),
  license_no    VARCHAR2(30)) TABLESPACE student_data;

ALTER TABLE students
  ADD CONSTRAINT pk_students PRIMARY KEY (student_id)
  USING INDEX TABLESPACE student_index;

ALTER TABLE students
  ADD CONSTRAINT uk_students_license
  UNIQUE (state, license_no)
  USING INDEX TABLESPACE student_index;

ALTER TABLE students
   ADD CONSTRAINT ck_students_st_lic
   CHECK ((state IS NULL AND license_no IS NULL) OR
          (state IS NOT NULL AND license_no is NOT NULL));

ALTER TABLE students
  ADD CONSTRAINT fk_students_state
  FOREIGN KEY (state) REFERENCES state_lookup (state);
```

Script 3–1. *Foreign Key with State Lookup DDL.*

The DDL script in Script 3–1 creates the STUDENTS table and table constraints. These constraints enforce the following rules:

Rule	Enforced With
A student is uniquely identified by a STUDENT_ID.	PRIMARY KEY constraint.
A student MAY have a driver's license. If they do, that state and license combination is unique among all other students.	UNIQUE constraint on STATE and LICENSE.

(*continued*)

Rule	Enforced With
If STATE is NULL then LICENSE_NO must be NULL; otherwise both must be NOT NULL.	CHECK constraint on STATE and LICENSE.
A student MAY have a column value for STATE. If they do, the STATE abbreviation is valid with respect to the STATE_LOOKUP table.	FOREIGN KEY constraint on STATE.

3.3.1 Four Types of Errors

The rules of referential integrity are enforced during updates and deletes on parent tables and inserts and updates on child tables. The SQL statements affected by referential integrity are:

PARENT-UPDATE	You cannot UPDATE a STATE in STATE_LOOKUP with a value so as to leave students with a state abbreviation that is no longer in the STATE_LOOKUP table.
PARENT-DELETE	You cannot DELETE a STATE and leave students with a state that is no longer in the parent lookup table. For example, if there are students with a California license, which use the abbreviation 'CA,' you cannot delete the 'CA' row from STATE_LOOKUP.
CHILD-INSERT	You cannot INSERT a student with a state that is not found in the STATE_LOOKUP table. For example, you cannot insert a student with a license and set the STATE column to a value not found in the STATE_LOOKUP table.
CHILD-UPDATE	You cannot UPDATE a student and replace the state with a state not found in the parent state lookup table.

The following SQL statements demonstrate the four error types and the Oracle error returned when the constraint is violated. These inserts, updates, and deletes behave assuming the data for the STATE_LOOKUP and STUDENTS tables:

STATE_LOOKUP

State	State Description
CA	California
NY	New York
NC	North Carolina

STUDENTS

Student ID	Student Name	College Major	Status	State	License NO
A101	John	Biology	Degree	NULL	NULL
A102	Mary	Math/Science	Degree	NULL	NULL
A103	Kathryn	History	Degree	CA	MV-232-13
A104	Steven	Biology	Degree	NY	MV-232-14
A105	William	English	Degree	NC	MV-232-15

The first two SQL statements are changes to the parent table. Prior to changing the parent, Oracle must examine the contents of the child table to ensure data integrity.

PARENT-UPDATE

```
SQL> UPDATE state_lookup
  2  SET state = 'XX'
  3  WHERE state = 'CA';

UPDATE state_lookup
*
ERROR at line 1:
ORA-02292: integrity constraint (SCOTT.FK_STUDENTS_STATE)
violated - child record found
```

PARENT-DELETE

```
SQL> DELETE FROM state_lookup
     2  WHERE state = 'CA';

DELETE FROM state_lookup
*
ERROR at line 1:
ORA-02292: integrity constraint (SCOTT.FK_STUDENTS_STATE)
violated - child record found
```

The next two statements are changes to the child table. Each DML on the child requires that Oracle examine the contents of the parent to ensure data integrity.

CHILD-INSERT

```
SQL> INSERT INTO STUDENTS
  2  VALUES ('A000',
  3  'Joseph','History','Degree','XX','MV-232-00');

INSERT INTO STUDENTS
*
ERROR at line 1:
ORA-02291: integrity constraint (SCOTT.FK_STUDENTS_STATE)
violated - parent key not found
```

CHILD-UPDATE

```
SQL> UPDATE students
  2  SET state = 'XX'
  3  WHERE student_id = 'A103';

UPDATE students
*
ERROR at line 1:
ORA-02291: integrity constraint (SCOTT.FK_STUDENTS_STATE)
violated - parent key not found
```

For Oracle to enforce these four rules, it must read from both tables. A simple INSERT into a STUDENTS table, given a foreign key constraint, requires that Oracle read the STATE_LOOKUP table. If we DELETE from the STATE_LOOKUP table, then Oracle must read the STUDENTS table to first ensure there is no student row that contains a STATE value that references the STATE_LOOKUP row.

In each of the four types of errors above, the error number is the same: ORA-02291.

Referential integrity is a critical part of a database design. It is rare for a table to not be either a parent or child of some other table. If you ever look at a data model printed on a large scale graphics plotter, the first thing you will notice are tables with no lines—no parents and no children.

3.3.2 Delete Cascade

You have the option within the foreign key syntax to specify a delete cascade feature. This feature only affects delete statements in the parent table.

With this option, a delete from the parent will automatically delete all relevant children. Had we created the foreign key constraint in Script 3–1,

"Foreign Key," with the DELETE CASCADE option, then the following SQL would delete the record in the STATE_LOOKUP table for California plus all students that have a California license.

```
SQL> DELETE FROM state_lookup
     2  WHERE state = 'CA';
```

The DELETE CASCADE syntax is:

```
ON DELETE CASCADE
```

The syntax for the foreign constraint, shown in Script 3–1, can be rewritten with the cascade option as follows:

```
ALTER TABLE students
  ADD CONSTRAINT fk_students_state
  FOREIGN KEY (state) REFERENCES state_lookup (state)
  ON DELETE CASCADE;
```

The delete cascade option should be the exception rather than the rule. Lots of data can be inadvertently lost with an accidental delete of a row in the parent lookup table. There are applications where this option is very useful. If data is temporary—only lives for a short time and is eventually deleted—then this is very convenient.

Delete cascade can span multiple tables. A parent can cascade to a child and that child can cause a delete cascade to other tables. If there is a foreign key constraint in the chain, without the cascade option, the delete from the parent fails.

Deletes that cascade over several tables can potentially affect other parts of the system. A lengthy delete that spans tables and deletes millions of rows will require comparable rollback space to write undo information. Rollback space should be analyzed if the cascade delete is excessive. Additionally, performance of concurrent queries against the tables will be affected. Consider the following when declaring a delete cascade option.

- Does the cascade fit the application? An accidental delete from a parent look table should not delete customer accounts.
- What is the chain being declared? Look at what tables cascade to other tables. Consider the potential impact and magnitude of a delete and how it would impact performance.

3.3.3 Mandatory Foreign Key Columns

The foreign key constraints stipulates the rule that a child MAY be a member of the parent. If the child is a member then there must be integrity. A NOT NULL constraint on the foreign key replaces MAY with MUST.

The foreign key constraint in Script 3–1 enforces the rule.

- A student MAY have a driver's license. If they do, that state is a member of STATE_LOOKUP.

This rule can have a different flavor—restated here:

- A student MUST have a driver's license and the state is a member of STATE_LOOKUP.

The latter is still a statement of referential integrity; there still exists a one-to-many relationship. The rule changes to MUST when the NOT NULL constraint is declared on the child column. The business rules are revised. Because the STATE and LICENSE are mandatory, the CHECK constraint is removed. The foreign key relationship changes from MAY to MUST.

Script 3–2 DDL for the STUDENTS and STATE_LOOKUP tables enforces the following:

Rule	**Enforced With**
A student is uniquely identified by a STUDENT_ID.	PRIMARY KEY constraint
A student MUST have a driver's license. The state and license combination is unique among all other students.	UNIQUE constraint, NOT NULL on STATE, NOT NULL on LICENSE
A student MUST have a column value for STATE. The STATE abbreviation is valid with respect to the STATE_LOOKUP table.	FOREIGN KEY constraint, NOT NULL on STATE

The DDL to enforce these rules is similar to Script 3–1. There is no CHECK constraint and NOT NULL constraints are added. There is no change to the STATE_LOOKUP table to accommodate the rule changes.

```
CREATE TABLE students
 (student_id    VARCHAR2(10) NOT NULL,
  student_name  VARCHAR2(30) NOT NULL,
  college_major VARCHAR2(15) NOT NULL,
  status        VARCHAR2(20) NOT NULL,
  state         VARCHAR2(2)  NOT NULL,
```

```
  license_no    VARCHAR2(30) NOT NULL)
TABLESPACE student_data;

ALTER TABLE students
  ADD CONSTRAINT pk_students PRIMARY KEY (student_id)
  USING INDEX TABLESPACE student_index;

ALTER TABLE students
  ADD CONSTRAINT uk_students_license
  UNIQUE (state, license_no)
  USING INDEX TABLESPACE student_index;

ALTER TABLE students
  ADD CONSTRAINT fk_students_state
  FOREIGN KEY (state) REFERENCES state_lookup (state);
```

Script 3–2. *Foreign Key with Mandatory Constraints.*

3.3.4 Referencing the Parent Syntax

When you create a foreign key constraint, the column(s) on that foreign key reference the column(s) that make up a PRIMARY KEY or UNIQUE constraint. In the case where a foreign key references a parent primary key, the column does not need to be specified. For example, Script 3–2 uses the following syntax:

```
ALTER TABLE students
  ADD CONSTRAINT fk_students_state
  FOREIGN KEY (state) REFERENCES state_lookup (state);
```

When no parent column is referenced, the default referenced column is the primary key of the parent. The following would work just as well.

```
ALTER TABLE students
  ADD CONSTRAINT fk_students_state
  FOREIGN KEY (state) REFERENCES state_lookup;
```

When your foreign key references a parent column with a unique constraint, that parent column must be specified in the ADD CONSTRAINT statement.

3.3.5 Referential Integrity across Schemas and Databases

A foreign key in a table can refer to a parent table in another schema. This is not desirable, at least from a database administrator's perspective. An application that sits wholly within a single schema is very portable. The

entire application that exports as a single owner easily imports into another database. When functionality spans schemas, the migration is not such a smooth process. Often, the application is not fully ported because the functionality in the other schemas is overlooked in the port. An export dump file will import with errors. The foreign key constraints will fail because the referenced schema does not exist, the object in the referenced schema does not exist, or privileges from the referenced schema have not been created.

Referential integrity can be implemented, to various degrees, across databases. This can be complicated. Options are to use triggers, refreshed materialized views, or Oracle replication. Materialized views are a straightforward and balanced solution. The materialized view becomes an object in the local schema. It is refreshed using data from another database. A database link is required along with the CREATE ANY MATERIALIZED VIEW privilege. The refresh rate of the view does not make the data real-time but can be near real-time if the materialized view is refreshed frequently.

Assume that the students' database needs relatively fresh data from a government driver's license agency. The students' database needs data from a LICENSE table. In this context the license agency is the database and the LICENSE table is a master table.

The LICENSE table exists on the government agency server, AGENCY.GOV, in a database name PROD, with a username/password of SCOTT/TIGER. The LICENSE table must have a primary key. The objective is to have a local snapshot of the LICENSE table that is refreshed once every 15 minutes. This is not real-time but certainly satisfies the needs of the student's database. The remote connectivity is illustrated in Figure 3–3.

The process starts by creating a Materialized View Log of the LICENSE table on the PROD.AGENCY.GOV database. This log is used by Oracle to track changes to the master table. This data in log format is used to refresh the materialized view in the student's database.

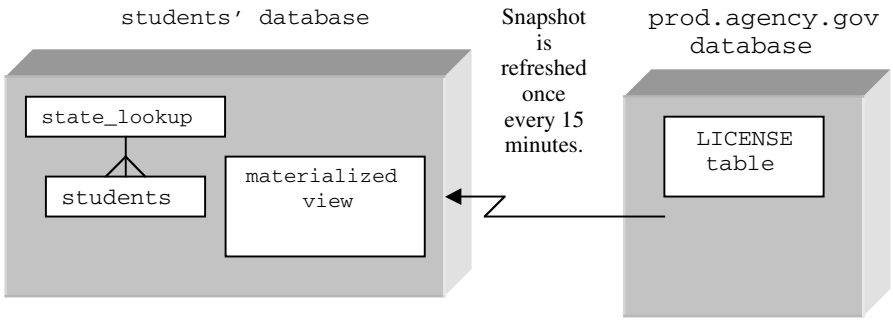

Figure 3–3 *Materialized View.*

On the remote database:

```
CREATE MATERIALIZED VIEW LOG ON LICENSE;
```

On the students' database, create a link to the remote PROD database; then create the materialized view. This view will be refreshed once every 15 minutes. If the network is down, the student's application still has access to the most recent refresh.

```
CREATE DATABASE LINK prod.agency.gov
  CONNECT TO scott IDENTIFIED BY tiger
  USING 'prod.agency.gov';

CREATE MATERIALIZED VIEW STUDENT_LICENSE_RECORDS
 REFRESH FAST NEXT SYSDATE + 1/96
 AS SELECT * FROM licenses@prod.agency.gov
```

The students' tables can now reference a local snapshot of data from the government database table with license information.

Creating materialized views requires non-default privileges. The materialized view should be created with storage clauses based on the size of the snapshot.

3.3.6 Multiple Parents and DDL Migration

A child table frequently has multiple lookup tables. The use of lookup tables greatly enhances the overall integrity of the data. The STUDENTS table began with a two-character column STATE. Integrity was added to that column by creating a lookup table, STATE_LOOKUP. Once the STATE_LOOKUP table is populated, the foreign key constraint can be created. The domain of STATE abbreviations and state descriptions can grow without impact to child tables. Lookup tables are often added throughout the development process as a means to improve the integrity of the data.

From an end user's perspective, maintenance of lookup tables usually addresses the subject of end user roles and privileges. For any application there are specific roles and privileges to run the application. However, changes to tables such as STATE_LOOKUP would, and should, require special access.

Developing the application software to support end user maintenance of lookup tables is mostly cut-and-paste. Most lookup tables have two columns. Once the first lookup maintenance screen is developed, maintenance for other lookup tables is somewhat repetitious.

Application code to display lookup data follows a standard practice of never showing the primary key column. To display student data on a screen requires joining the STUDENTS and STATE_LOOKUP tables. For each student, the student name and state description are shown but not the two character

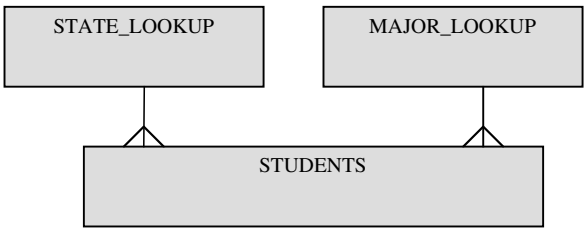

Figure 3–4 *Multiple Parents.*

state field. HTML form elements such as drop down lists are populated with the state description, but by using the state as the option value.

```
<OPTION VALUE=state>state_description</OPTION>
```

An observation of the data outlined in the tables on p. 108 might lead one to believe that student college majors should be in a lookup table as well. This additional lookup table would restrict the domain of college major descriptions to the values controlled through a lookup table. The MAJOR_LOOKUP table would store the descriptions like "Biology" and "Math." This addition would be modeled with the entity diagram shown in Figure 3–4.

The addition of a MAJOR_LOOKUP table to a STUDENTS and STATE_LOOKUP, illustrated in Figure 3–2, is accomplished with the following. Assume there is data present in the STATE_LOOKUP and STUDENTS table as outlined in on p. 108.

The lookup table, MAJOR_LOOKUP, must be created and populated. The following includes the CREATE script, primary key, and data load.

```
CREATE TABLE major_lookup
 (major      VARCHAR2(2)  NOT NULL,
  major_desc VARCHAR2(15) NOT NULL)
TABLESPACE student_data;

INSERT INTO major_lookup values ('UD','Undeclared');
INSERT INTO major_lookup values ('BI','Biology');
INSERT INTO major_lookup values ('MS','Math/Science');
INSERT INTO major_lookup values ('HI','History');
INSERT INTO major_lookup values ('EN','English');

ALTER TABLE major_lookup
  ADD CONSTRAINT pk_major_lookup PRIMARY KEY (major)
  USING INDEX TABLESPACE student_index;
```

The STUDENTS table must be changed. It stores the college major as VARCHAR(15). This must be replaced with a two-character field that will be a

foreign key to the MAJOR_LOOKUP table. This approach creates a temporary copy (table TEMP) of the STUDENTS table, including the data. The STUDENTS table is dropped; a new STUDENTS table is created and the saved data is migrated back into the new STUDENTS table.

```
CREATE TABLE TEMP AS SELECT * FROM STUDENTS;

DROP TABLE STUDENTS;

CREATE TABLE students
 (student_id      VARCHAR2(10) NOT NULL,
  student_name    VARCHAR2(30) NOT NULL,
  college_major   VARCHAR2(2)  NOT NULL,
  status          VARCHAR2(15) NOT NULL,
  state           VARCHAR2(2),
  license_no      VARCHAR2(30))
TABLESPACE student_data;

INSERT INTO STUDENTS
SELECT student_id,
       student_name,
       decode
       (  college_major,
         'Undeclared'   , 'UD',
         'Biology'      , 'BI',
         'Math/Science' , 'MS',
         'History'      , 'HI',
         'English'      , 'EN'
       ),
       status,
       state,
       license_no
FROM temp;
```

The new STUDENTS table is populated, but with BI for Biology. The constraints are added to the STUDENTS table. This includes:

1. PRIMARY KEY
2. UNIQUE constraint on STATE, LICENSE No
3. CHECK constraint on STATE and License No
4. Foreign Key to STATE_LOOKUP
5. Foreign Key to MAJOR_LOOKUP

```
1. ALTER TABLE students
   ADD CONSTRAINT pk_students PRIMARY KEY (student_id)
   USING INDEX TABLESPACE student_index;

2. ALTER TABLE students
```

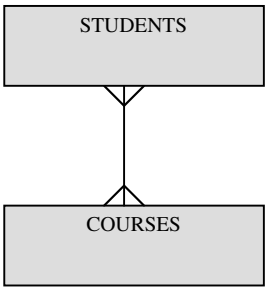

Figure 3–5 *Many-to-Many Relationship.*

```
   ADD CONSTRAINT uk_students_license
   UNIQUE (state, license_no)
   USING INDEX TABLESPACE student_index;

3. ALTER TABLE students
   ADD CONSTRAINT ck_students_st_lic
   CHECK ((state IS NULL AND license_no IS NULL) OR
          (state IS NOT NULL AND license_no is NOT NULL));

4. ALTER TABLE students
   ADD CONSTRAINT fk_students_state
   FOREIGN KEY (state) REFERENCES state_lookup;

5. ALTER TABLE students
   ADD CONSTRAINT fk_students_college_major
   FOREIGN KEY (college_major) REFERENCES major_lookup;
```

3.3.7 Many-to-Many Relationships

Data modeling tools will draw, at the logical level, a many-to-many relationship with the notation show in Figure 3–5.

The model in Figure 3–5 demonstrates that a student can take several courses, while each course is taught to more than one student. There is a many-to-many relationship between students and courses. Physically, this is not implemented directly; rather, we include a cross-reference table. That cross-reference table, STUDENTS_COURSES, will contain all the courses a student takes, plus all students that take a particular course. The physical model becomes the graph in Figure 3–6.

The following types of constraints are commonly applied with the physical implementation of a many-to-many relationship.

- We have a primary key in STUDENTS.
- We have a primary key in COURSES.

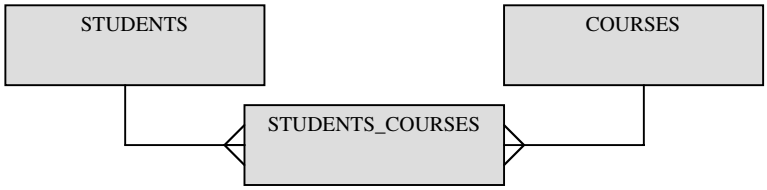

Figure 3–6 *Physical Many-to-Many Relationship.*

- We have a CONCATENATED PRIMARY KEY in the cross-reference table, STUDENTS_COURSES.
- Part of this primary key is a foreign key to the STUDENTS table; part of it is a foreign key to the COURSES table.
- We have a foreign key in STUDENTS_COURSES that ensures each student in that table is also a student in STUDENTS. The same column in the foreign key is part of the primary key.
- We have a foreign key in STUDENTS_COURSES that ensures each course in that table is also a course in COURSES. The same column in this foreign key is part of the primary key.

When we apply these rules in the form of constraints, we will have three CREATE TABLE statements, three PRIMARY KEYS, and two FOREIGN KEY constraints.

The table description for the COURSES table is shown here:

```
Name                           Null?    Type
------------------------------ -------- ------------
COURSE_NAME                             VARCHAR2(10)
COURSE_DESC                             VARCHAR2(20)
NO_OF_CREDITS                           NUMBER(2,1);
```

The table description for the STUDENTS_COURSES table is shown here:

```
Name                           Null?    Type
------------------------------ -------- ------------
STUDENT_ID                              VARCHAR2(10)
COURSE_NAME                             VARCHAR2(10)
```

The columns of the cross-reference table contain columns that must reference the parents, in this case, STUDENTS and COURSES. This is first step. Additional columns can be added to this table, such as the professor who teaches the class and when it is taught. A cross-reference table can be just the joining columns. It can also contain additional attributes.

The DDL for this many-to-many relationship is show in Script 3–3.

```
CREATE TABLE students
 (student_id    VARCHAR2(10) NOT NULL,
  student_name  VARCHAR2(30) NOT NULL,
  college_major VARCHAR2(15) NOT NULL,
  status        VARCHAR2(20) NOT NULL,
  state         VARCHAR2(2),
  license_no    VARCHAR2(30))
TABLESPACE student_data;

CREATE TABLE courses
 (course_name   VARCHAR2(10) NOT NULL,
  course_desc   VARCHAR2(20) NOT NULL,
  no_of_credits NUMBER(2,1)  NOT NULL)
TABLESPACE student_data;

CREATE TABLE students_courses
 (student_id    VARCHAR2(10) NOT NULL,
  course_name   VARCHAR2(10) NOT NULL)
TABLESPACE student_data;

ALTER TABLE students
  ADD CONSTRAINT pk_students
  PRIMARY KEY (student_id)
  USING INDEX TABLESPACE student_index;

ALTER TABLE courses
  ADD CONSTRAINT pk_courses
  PRIMARY KEY (course_name)
  USING INDEX TABLESPACE student_index;

ALTER TABLE students_courses
  ADD CONSTRAINT pk_students_courses
  PRIMARY KEY (student_id, course_name)
  USING INDEX TABLESPACE student_index;

ALTER TABLE students_courses
  ADD CONSTRAINT fk_students_courses_st_id
  FOREIGN KEY (student_id)
  REFERENCES students (student_id);

ALTER TABLE students_courses
  ADD CONSTRAINT fk_students_courses_course
  FOREIGN KEY (course_name)
  REFERENCES courses (course_name);
```

Script 3–3. *Many-to-Many Relationship.*

3.3.8 Self-Referential Integrity

Self-referential integrity is common in many applications. Self-referential integrity allows parent-child relationships to exist between instances of the same entity. For example, all professors are in the following PROFESSORS table, including the individual department heads.

Some professors have a department head. These department heads are also professors. The following creates a table and a primary key, and then establishes a self-referential integrity constraint.

```
CREATE TABLE professors
 (prof_name     VARCHAR2(10) NOT NULL,
  specialty     VARCHAR2(20) NOT NULL,
  hire_date     DATE         NOT NULL,
  salary        NUMBER(5)    NOT NULL,
  dept_head     VARCHAR2(10))
TABLESPACE student_data;

ALTER TABLE professors
  ADD CONSTRAINT pk_professors
  PRIMARY KEY (prof_name)
  USING INDEX TABLESPACE student_index;

ALTER TABLE professors
  ADD CONSTRAINT fk_professors_prof_name
  FOREIGN KEY (dept_head)
  REFERENCES professors (prof_name);
```

This permits a scenario where a professor MAY have a department head (DEPT_HEAD) and if they have a department head, that column value MUST be an existing PROF_NAME. Figure 3–7 shows, for example, that Blake is Milton's department head.

Self-Referential Integrity, for the aforementioned data, is one level deep. The foreign key definition permits unlimited nesting. Multilevel nesting is illustrated with the following example. We create a sample table TEMP with three columns: WORKER, SALARY, and MANAGER. A worker may or may not have a manager. A manager must first be inserted as a worker. Workers can be

```
PROF_NAME  SPECIALTY  HIRE_DATE     SALARY DEPT_HEAD
---------- ---------- --------- ---------- ----------
Blake      English    26-JUL-99      10000
Wilson     Physics    27-JUL-98      10000
Milton     English    24-JUL-02      10000 Blake
Jones      Math       29-JUN-03      10000 Wilson
```

Figure 3–7 *Self-Referential Integrity Data.*

managers of other workers who may manage other workers. The relationships among these workers and their salaries is shown as well.

```
CREATE TABLE TEMP
   (worker   VARCHAR2(10) PRIMARY KEY,
    salary   NUMBER(3),
    manager  VARCHAR2(10) REFERENCES TEMP (worker));
```

Allen manages Bill and Beth. Beth manages Cindy and Carl. Carl manages Dean and Dave.

```
Allen (salary=10) manages: Bill (salary=10), Beth
(salary=10)
Beth  (salary=10) manages: Cindy (salary=5), Carl
(salary=5)
Carl  (salary=5)  manages: Dean  (salary=5), Dave
(salary=5)
```

This data contains multilevel relationships forming a logical tree organization. A SELECT statement using a CONNECT BY and START AT clause enables a query to return, for example, the sum of all salaries starting at a specific point in the tree. The inserts for this data are:

```
INSERT INTO TEMP values ('Allen',  10,  null  );
INSERT INTO TEMP values ('Bill' ,  10, 'Allen');
INSERT INTO TEMP values ('Beth' ,  10, 'Allen');
INSERT INTO TEMP values ('Cindy',  5,  'Beth' );
INSERT INTO TEMP values ('Carl' ,  5,  'Beth' );
INSERT INTO TEMP values ('Dean' ,  5,  'Carl' );
INSERT INTO TEMP values ('Dave' ,  5,  'Carl' );
```

The following is a START AT SELECT statement to return Beth's salary including all those who work for Beth. This is the sum of salaries for Beth, Cindy, and Carl, plus Dean and Dave who are managed by Carl: the sum is $30.00.

```
SQL> SELECT sum(salary)
  2  FROM temp
  3  START WITH worker='Beth'
  4  CONNECT BY PRIOR worker=manager;

SUM(SALARY)
-----------
         30
```

3.3.9 PL/SQL Error Handling with Parent/Child Tables

Foreign key constraint errors are captured with mapping an exception to the Oracle error, minus 2291. The following procedure inserts a student row and captures the duplicate inserts. In this case a duplicate could be a primary key constraint violation or a unique constraint error. Both generate the DUP_VAL_ON_INDEX exception.

The DDL in Script 3–1 declares a CHECK constraint on STATE and LICENSE NO. The following procedure inserts a student and captures duplicate inserts that may violate the primary key or unique constraint, foreign key constraint, and check constraint.

```
CREATE OR REPLACE PROCEDURE
    insert_student(v_student_id VARCHAR2,
        v_student_name VARCHAR2,
        v_college_major VARCHAR2, v_status VARCHAR2,
        v_state VARCHAR2, v_license_no VARCHAR2)
IS
    check_constraint_violation exception;
    pragma exception_init(check_constraint_violation, -2290);

    foreign_key_violation exception;
    pragma exception_init(foreign_key_violation, -2291);

BEGIN
    INSERT INTO students VALUES
        (v_student_id, v_student_name,
         v_college_major, v_status,
         v_state, v_license_no);
    dbms_output.put_line('insert complete');
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        dbms_output.put_line('PK or unique const violation');
    WHEN check_constraint_violation THEN
        dbms_output.put_line('check constraint violation');
    WHEN foreign_key_violation THEN
        dbms_output.put_line('foreign key violation');
END;
```

3.3.10 The Deferrable Option

In this section we use two tables that store generic parent/child data. The data model for this is shown in Figure 3–8.

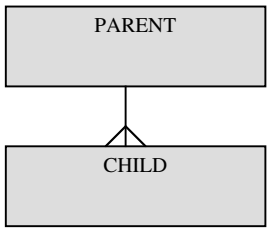

Figure 3–8 *Parent-Child Relationship.*

The PARENT table description is:

```
Name                           Null?    Type
------------------------------ -------- ------------
PARENT_NAME                             VARCHAR2(2)
PARENT_DESC                             VARCHAR2(10)
```

The CHILD table description is:

```
Name                           Null?    Type
------------------------------ -------- ------------
CHILD_NAME                              VARCHAR2(2)
PARENT_NAME                             VARCHAR2(10)
```

The DDL, shown next, includes a DEFERRABLE option on the foreign key constraint.

```
CREATE TABLE parent
 (parent_name  VARCHAR2(2) CONSTRAINT pk_parent PRIMARY
  KEY, parent_desc  VARCHAR2(10));

CREATE TABLE child
 (child_name  VARCHAR2(10),
  parent_name VARCHAR2(2));

ALTER TABLE child ADD CONSTRAINT fk_child_parent_name
  FOREIGN KEY (parent_name)
  REFERENCES parent (parent_name) DEFERRABLE;
```

This DEFERRABLE attribute means that we can choose to defer the constraint and load a set of data into the parent and child tables, without regard to referential integrity, under the assumption that when the load completes, the data will be clean. Then a commit will automatically apply the rule on our loaded data. We can load 10 records into a child table that has no parents and

then load the parents. Validation occurs on the commit. If the data we loaded violates the referential integrity rule, the transaction is rolled back.

You can do this in SQL*Plus with the following.

```
SET constraints ALL DEFERRED;
INSERT INTO child VALUES ('child_1','P1');
INSERT INTO child VALUES ('child_2','P1');
INSERT INTO child VALUES ('child_3','P2');
INSERT INTO child VALUES ('child_4','P3');
INSERT INTO child VALUES ('P1','a parent');
INSERT INTO child VALUES ('P2','a parent');
INSERT INTO child VALUES ('P3','a parent');
COMMIT;
```

You can use this functionality in a stored procedure with the following:

```
CREATE OR REPLACE PROCEDURE P IS
BEGIN
    EXECUTE IMMEDIATE 'SET constraints ALL DEFERRED';

    INSERT INTO child VALUES ('child_1','P1');
    INSERT INTO child VALUES ('child_2','P1');
    INSERT INTO child VALUES ('child_3','P2');
    INSERT INTO child VALUES ('child_4','P4');
    INSERT INTO child VALUES ('P1','a parent');
    INSERT INTO child VALUES ('P2','a parent');
    INSERT INTO child VALUES ('P3','a parent');
    COMMIT;
END P;
```

The general motivation for this is that the data comes in an inconsistent order. The aforementioned procedure inserts all child rows, then the parents. The aforementioned inserts contain hard-coded literal values. A more realistic situation could involve records read from a file using the UTL_FILE package. The input file would contain many rows out of order; that is, all child records followed by all parent records. In this case a procedure could contain a simple loop, iterate over all child inserts followed by all parent inserts, and then perform integrity checking upon a commit. In very select situations this can be a reasonable approach.

In general, there are other tools available to obviate the need for this option.

- You can write an exception handler around each child insert and should that fail, because there is no parent, you write code in the exception handler to insert a parent. Following the parent insert, you insert the child. For example, a child insert would be enclosed within an exception handler block similar to the following block,

which maps the foreign key constraint violation (ORA-02291) to an exception. Upon an exception, insert a parent and then the child.

```
DECLARE
    no_parent EXCEPTION;
    PRAGMA EXCEPTION_INIT (no_parent, -2291);
    new_parent VARCHAR2 (2) := 'P6';
    new_child VARCHAR2(10) := 'child_5';
BEGIN
    INSERT INTO child VALUES (new_child,
         new_parent);
EXCEPTION
    WHEN no_parent THEN
        INSERT INTO parent VALUES (new_parent,
             'no desc');
        INSERT INTO child VALUES (new_child,
             new_parent);
END;
```

- You can store failed child records in a PL/SQL table and when you complete inserting all parents, you go back and load the child records saved in the PL/SQL table. The PL/SQL table would be derived from the child table with the following syntax:

```
TYPE temporary_table_type IS TABLE OF
    CHILD%ROWTYPE
     INDEX BY BINARY_INTEGER;
temporary_table temporary_table_type;
```

- Another option is to use a temporary table to store failed child records. This is a solution similar to using a PL/SQL Table. The PL/SQL table is manipulated like an array. The temporary table is accessed through SQL. With this solution, you load child records into this temporary repository, load all parents, then load from the temporary table into the final child table. The SQL for a temporary table is:

```
CREATE GLOBAL TEMPORARY TABLE
CHILD_TEMP
 (child_name  VARCHAR2(10),
  parent_name VARCHAR2(2))ON COMMIT DELETE ROWS;
```

The deferrable option is a powerful tool but should not be used as a standard practice because it disables the very rules that are strongly encouraged—even if the deferred state is a temporary one.

The SQL*Plus script and aforementioned stored procedure set the constraint to a deferred state with the statement:

```
SET CONSTRAINT ALL DEFERRED;
```

This SET command is one option for deferring the constraint. The other option is to replace ALL with the specific constraint name—that would be FK_CHILD_PARENT_NAME. This means your application code specifically references a constraint name. The SET CONSTRAINT ALL only affects your current transaction; the code is more generic because it does not specifically reference a constraint name. As a matter of style and preference, specific mentioning of constraint names in the code is not a recommendation.

When we create a constraint, using the following DDL, it means that we have the option to write code, or use SQL*Plus, to temporarily defer that constraint. Unless we DEFER the constraint, everything remains the same. But, if we choose to write code that temporarily breaks the rule, we must first DEFER the constraint.

```
ALTER TABLE child ADD CONSTRAINT fk_child
  FOREIGN KEY (parent_name)
  REFERENCES parent (parent_name) DEFERRABLE;
```

An option to this is to initially create the constraint in a deferred state. The DDL for this is the following.

```
ALTER TABLE child ADD CONSTRAINT fk_child
  FOREIGN KEY (parent_name)
  REFERENCES parent (parent_name) DEFERRABLE
  INITIALLY DEFERRED;
```

With this option everything is reversed. When we load data into the child and then parent tables, the constraint is deferred—this is the same as not having a constraint. Should we want to write code with the constraint enforced, as we perform each insert, then we would precede those insert statements with:

```
SET CONSTRAINT ALL IMMEDIATE;
```

3.4 Check

Declaring a database column to store a person's age starts with the following:

```
CREATE TABLE temp (age NUMBER);
```

This command will work, but the range of the data type far surpasses the domain of a person's age. The goal is to restrict one's age to the range: 1 to 125—any value outside that range is rejected. A dimension on the datatype can impose a restriction on the column so that any value, outside a three-digit number, is invalid data.

```
CREATE TABLE temp (age NUMBER(3));
```

A dimension scales down the range of valid values. Still, values far greater than 125 can be inserted—any three-digit number is possible. An age of 999 is not acceptable. In general, a CHECK constraint is used to restrict the data to a real world domain. To restrict values to integer values between 1 and 125, create a check constraint on the column.

```
CREATE TABLE temp (age NUMBER(3));

ALTER TABLE temp ADD CONSTRAINT ck_temp_age CHECK
    ((AGE>0) AND (AGE <= 125));
```

Now, the set of values we are able to insert into AGE is the set (1, 2, 125). This matches our real-world domain, with one exception, a NULL insert.

The aforementioned CREATE and ALTER TABLE statements permit a NULL. That may be within the business rules. Maybe the database does not store an age for every person. If this is the case then the aforementioned DDL is acceptable and enforces the rule.

To restrict the aforementioned AGE column to the 1–125 range and not permit nulls, attach a NOT NULL constraint. The DDL with the NOT NULL enforcement is now:

```
CREATE TABLE temp (age NUMBER(3) NOT NULL);

ALTER TABLE temp ADD CONSTRAINT ck_temp_age CHECK
  ((AGE>0) AND (AGE<=125));
```

The response from a CHECK constraint violation is an ORA error with the -2290 error number.

```
SQL> insert into temp values (130);
insert into temp values (130)
*
ORA-02290: check constraint (SCOTT.CK_TEMP_AGE) violated.
```

When a row is inserted or updated and there is a check constraint, Oracle evaluates the check constraint as a Boolean expression. For the aforementioned check, the row is inserted provided the expression "the AGE is within the (1,125) range" evaluates to true. The row is rejected if it is not TRUE.

The CHECK constraint does not have to be a continuous range. Suppose we want to constrain a column value to the following boundaries.

```
(NOT NULL) AND (range in 0-10 OR 999 OR 9999)
```

The check constraint for this would be the following.

```
CREATE TABLE temp (a NUMBER);
ALTER TABLE temp ADD CONSTRAINT ck_temp_a CHECK
  (((a>=0) AND (a<=10)) OR (a=999) OR (a=9999));
```

Oracle does not evaluate the logic of the constraint defined in the DDL statement. The following constraint will never allow you to insert a row but you will certainly have no trouble creating it:

```
ALTER TABLE temp ADD CONSTRAINT ck_temp_age CHECK
  ((AGE<0) AND (AGE>=125));
```

Check constraints can be used to implement a Boolean constraint in a database column. Some databases have a Boolean type; there is no BOOLEAN table column type in Oracle—there is a BOOLEAN datatype in PL/SQL. To simulate a Boolean column use a check constraint:

```
CREATE TABLE temp(enabled NUMBER(1) NOT NULL);

ALTER TABLE temp ADD CONSTRAINT ck_temp_enabled CHECK
  (enabled IN (0, 1));
```

You can use a VARCHAR2 as well. The following is another approach:

```
CREATE TABLE temp(enabled VARCHAR2(1) NOT NULL);

ALTER TABLE temp ADD CONSTRAINT ck_temp_enabled CHECK
  (enabled IN ('T', 'F', 't', 'f'));
```

You can restrict a column to a discrete set of character string values. The following uses a check constraint to limit the values of a status field: 'RECEIVED,' 'APPROVED,' 'WAITING APPROVAL.'

```
CREATE TABLE temp(status VARCHAR2(16) NOT NULL);

ALTER TABLE temp ADD CONSTRAINT ck_temp_status CHECK
 (status IN
 ('RECEIVED','APPROVED','WAITING APPROVAL'));
```

3.4.1 Multicolumn Constraint

A CHECK constraint can be a composite of several columns. The following table stores the dimensions of a box. We want to restrict each dimension of the box to a range within 1 and 10, plus the volume of the box must be less than 100. This means a range constraint on each column plus a constraint on the product of the dimensions.

```
CREATE TABLE box
 (length NUMBER(2) NOT NULL,
  width  NUMBER(2) NOT NULL,
  height NUMBER(2) NOT NULL);

ALTER TABLE box ADD CONSTRAINT ck_box_volume CHECK
  ((length*width*height<100) AND
   (length >  0) AND (length <= 10) AND
   (width  >  0) AND (width  <= 10) AND
   (height >  0) AND (height <= 10));
```

An insert of a zero dimension or a combination of dimensions that exceed the approved volume will generate the same constraint error. Both of the following INSERTS fail.

```
insert into box values (0,2,3);
insert into box values (8,8,8);
```

The error from each insert will be the following ORA constraint error:

```
ORA-02290: check constraint (SCOTT.CK_BOX_DIMENSION)
violated.
```

You can declare multiple constraints with different names. This accomplishes the same goal of enforcing a single constraint. The only advantage is that one can see, more specifically, the exact nature of the error based on the constraint name. The following declares a separate constraint for each range constraint and one final constraint for the volume restriction.

```
CREATE TABLE box
 (length NUMBER(2) NOT NULL,
  width  NUMBER(2) NOT NULL,
  height NUMBER(2) NOT NULL);

ALTER TABLE box ADD CONSTRAINT ck_box_length CHECK
  ((length > 0) AND (length <= 10));

ALTER TABLE box ADD CONSTRAINT ck_box_width CHECK
  ((width > 0) AND (width <= 10));

ALTER TABLE box ADD CONSTRAINT ck_box_height CHECK
  ((height > 0) AND (height <= 10));

ALTER TABLE box ADD CONSTRAINT ck_box_dimension CHECK
 ((length*width*height<100));
```

The same insert statements, used earlier, still fail, but the constraint name is more specific because each constraint enforces one part of the overall rule. Repeating the inserts:

```
insert into box values (0,2,3);
insert into box values (8,8,8);
```

gives the following errors from SQL*Plus:

```
ORA-02290: check constraint (SCOTT.CK_BOX_LENGTH)
violated.

ORA-02290: check constraint (SCOTT.CK_BOX_DIMENSION)
violated.
```

The difference between declaring a single constraint to enforce an overall rule or separate individual constraints is a style issue.

3.4.2 Supplementing Unique Constraints

Check constraints can be used to enforce a multicolumn NOT NULL constraint that stipulates that the columns are both NULL or both NOT NULL. This is a common practice with concatenated UNIQUE constraints. A table that stores student information will have a primary key but other columns may have a unique constraint; for example, a student driver's license—this information consists of a state abbreviation and license number. A student may not have a license; in this case both columns are null. If a student has a license, then both columns are NOT NULL and are further governed by a UNIQUE constraint.

A UNIQUE constraint enforces uniqueness among all NOT NULL values. For a concatenated UNIQUE constraint, one column can be NULL; the other column may not be NULL. We may not want this condition. The rule we want to enforce is:

(both columns are NULL) OR (both columns are NOT NULL)

To stipulate the type of constraint combine a CHECK constraint with the UNIQUE constraint.

```
CREATE TABLE temp (pk NUMBER PRIMARY KEY, a NUMBER, b NUMBER);
ALTER TABLE temp
   ADD CONSTRAINT uk_temp_a_b UNIQUE (a, b);
ALTER TABLE temp ADD CONSTRAINT ck_temp_a_b
CHECK ((a IS NULL AND b IS NULL) OR
       (a IS NOT NULL AND b is NOT NULL));
```

Given the aforementioned DDL, the following inserts will violate the check constraint:

```
INSERT INTO temp VALUES (6, 1, NULL);
INSERT INTO temp VALUES (7, NULL, 1);
```

3.4.3 Students Table Example

The following DDL illustrates the motivation for a check-unique constraint combination. The STUDENTS create-table DDL below does not require license information for a student—all other columns are mandatory, but STATE and LICENSE_NO can be NULL. The STATE and LICENSE_NO, if they exist, must be unique—this is the UNIQUE constraint. The CHECK constraint enforcement is: there can never be a STATE value with a NULL LICENSE NO or a LICENSE NO value with a NULL STATE.

```
CREATE TABLE students
 (student_id      VARCHAR2(10) NOT NULL,
  student_name    VARCHAR2(30) NOT NULL,
  college_major   VARCHAR2(15) NOT NULL,
  status          VARCHAR2(20) NOT NULL,
  state           VARCHAR2(2),
  license_no      VARCHAR2(30));

ALTER TABLE students
   ADD CONSTRAINT pk_students PRIMARY KEY (student_id)
   USING INDEX TABLESPACE student_index;

ALTER TABLE students
   ADD CONSTRAINT uk_students_license
   UNIQUE (state, license_no)
   USING INDEX TABLESPACE student_index;
ALTER TABLE students ADD CONSTRAINT ck_students_st_lic
CHECK ((state IS NULL AND license_no IS NULL) OR
       (state IS NOT NULL AND license_no is NOT NULL));
```

3.4.4 Lookup Tables versus Check Constraints

Compared to using a lookup table, the following check constraint has disadvantages.

```
ALTER TABLE temp ADD CONSTRAINT ck_temp_status CHECK
 (status IN
 ('RECEIVED','APPROVED','WAITING APPROVAL'));
```

You cannot extend the list of values without dropping and recreating the constraint. You cannot write application code that will show the list of possible values, without duplicating the list within the application. The lookup

table approach allows easy update and extension of the value set—that's not the case with the aforementioned check constraint.

On the other hand, to restrict person's age to a range from 1 to 125 with a lookup table is not practical. For a range-numeric rule the best choice is a check constraint.

3.4.5 Cardinality

When check constraints are used to restrict a column to a limited set of values, we may have a potential candidate for a bit map index. This only applies to those cases where the constraint is used to limit the column to a small number of possible values. Check constraints, other than numeric checks, are often columns with low cardinality. For example, consider a check constraint that restricts a column to a person's gender: 'M' or 'F' with the following:

```
CREATE TABLE temp(gender VARCHAR2(1) NOT NULL);

ALTER TABLE temp ADD CONSTRAINT ck_temp_gender CHECK
    (gender IN ('M', 'F'));
```

The column is a candidate for a bit map index if we frequently query this table based on gender. Bit map indexes are useful when the cardinality of the column is low (i.e., has only a few possible values). The following creates a bit map index on column GENDER.

```
CREATE BITMAP INDEX b_temp_gender ON TEMP
  (gender) TABLESPACE student_index;
```

3.4.6 Designing for Check Constraints

The pursuit of what columns need check constraints is sometimes overlooked. When developing new applications or migrating legacy systems to an Oracle database, the domain of an attribute often remains the same as the range of the data type. With this approach we would have permitted, in our earlier example, the insertion of someone's age to be 999 years. One reason for few check constraints in a database is simply the fact that no one knows the data well enough to state emphatically that there is some specific range of values. This is understandable.

Given this, you have two choices. Skip check constraints, or make some assumptions and the worst that can happen is that a record fails on a check constraint. In the latter case, you can always drop the constraint and recreate with the new rule, then rerun the application for which the insert failed.

Earlier we created a check constraint named CK_BOX_LENGTH. This constraint can be redefined with the following.

```
SQL> ALTER TABLE box DROP CONSTRAINT ck_box_length;

Table altered.

SQL> ALTER TABLE box ADD CONSTRAINT ck_box_length CHECK
  2      ((length > 0) AND (length <= 12));

Table altered.
```

Another reason for few check constraints appears to be the powerful enforcement capability in the client application. JavaScript in a browser is best for verifying that a field is not blank or not zero before sending the string over the network, only to have it rejected by a constraint error. It makes a lot of sense to enforce these types of rules in the client because the end user does not have to wait to realize they made an error on data entry. However, as illustrated in Figure 3–3, data goes into the database from multiple sources—not just end users and regardless of client-side rule enforcement, the rules of data integrity should still be in the database. Constraint checks will preserve data integrity for SQL*Plus operations, PL/SQL programs that may run nightly loads, and SQL*Loader runs that bring in data from other systems.

3.5 NOT NULL Constraints

The NOT NULL constraint is often referred to as a "mandatory" constraint (i.e., we say the AGE column is a mandatory value). You see this term sometimes in data modeling tools when you are asked what types of constraints you want to apply to a column.

The most common syntax for a not null constraint is to append NOT NULL within the column definition of the table. You can name a not null constraint. If you are hand-coding a lot of DDL, you may choose not to name each constraint—that can be a time-consuming task. Some tools, like Oracle Designer, automatically generate constraint names for you. This is a nice feature and should be used. The naming of constraints has been emphasized, rightfully so, within this chapter. However, naming NOT NULL constraints is just not as common as naming other constraints. In summary, it doesn't hurt to name them. The following illustrates some options.

```
CREATE TABLE temp(id NUMBER(1) NOT NULL);

CREATE TABLE temp(id NUMBER(1) CONSTRAINT nn_temp_id NOT
NULL);
```

There is one significant reason why we name PRIMARY KEY, UNIQUE, FOREIGN KEY, and CHECK constraints:

The Oracle constraint violation error message includes the name of the constraint. This is very helpful and allows us to quickly isolate the problem without having to look in the data dictionary or investigate the code that caused the error.

Oracle does not reference the constraint name in a NOT NULL constraint violation. It doesn't have to because the column name is included in the message. The following illustrates a table with three NOT NULL columns, but imagine this scenario with 20 columns. A PL/SQL code block is used to populate the table, but with one variable reset to NULL. The Oracle error message generated specifically identifies the column constraint violated. First the table:

```
CREATE TABLE temp
 (N1 NUMBER constraint nn_1 NOT NULL,
  N2 NUMBER constraint nn_2 NOT NULL,
  N3 NUMBER constraint nn_3 NOT NULL);
```

This is a short block, but it could just as well be several hundred lines of PL/SQL. The constraint error message identifies the column, saving us from isolating that part of the problem.

```
DECLARE
    v1 NUMBER := 1;
    v2 NUMBER := 2;
    v3 NUMBER := 3;
BEGIN
    v2 := null;
    INSERT INTO temp VALUES (v1, v2, v3);
END;
```

The result of running the aforementioned PL/SQL block is the following Oracle error—notice the column name, N2.

```
ORA-01400: cannot insert NULL into ("SCOTT"."TEMP"."N2")
```

So, naming the constraint served little purpose. The error says we tried to insert a NULL into the column N2. We only have to look into the PL/SQL code and trace how that variable was set to NULL.

3.6 Default Values

Consider the following table with its two columns:

```
CREATE TABLE TEMP (id NUMBER, value NUMBER);
```

The first two INSERT statements are identical. This third inserts a NULL for the column VALUE.

```
INSERT INTO temp             VALUES (1, 100);
INSERT INTO temp (id, value) VALUES (1, 100);
INSERT INTO temp (id)        VALUES (2);
```

Is this what you want? Do you want NULL? Maybe a zero would be better. The following replaces the DEFAULT NULL with a zero.

```
CREATE TABLE TEMP (id NUMBER,
                   value VARCHAR2(10) DEFAULT 0);
```

The default for VALUE is no longer NULL. This inserts a (2, 0)

```
INSERT INTO temp (id) VALUES (2);
```

If a zero can be interpreted as no data, then use a DEFAULT as was done earlier. There are situations where a zero equates to no data. One example is a checking account. If there is a zero balance, there is no money. Other situations require a NULL. Consider an environmental biologist measuring microbes in a polluted river. On some days, the microbe count may be zero. On other days the river may be frozen. On these days an insert should be a NULL. A zero value, on days when the river is frozen, could skew any analysis of the data. For the environmentalist, there is a difference between a zero and a NULL. A NULL means no sample was taken. For the checking account, a zero is equivalent to the absence of money.

PL/SQL expressions that include NULL values can be tricky. Refer to Chapter 11, Section 11.4 for additional discussion on this topic.

3.7 Modifying Constraints

You can disable a constraint with the syntax:

```
ALTER TABLE table_name DISABLE CONSTRAINT constraint_name;
```

This leaves the constraint defined in the data dictionary. It is just not being enforced. You might do this on a development database where you need to load data that you expect will not conform to the business rules. Once the data is loaded you view it with SQL*Plus. If the data looks good you can try to enable the constraint with:

```
ALTER TABLE table_name ENABLE CONSTRAINT constraint_name;
```

If the data is bad, the ENABLE command will fail. The Oracle error message will indicate that the constraint cannot be enforced because the data does not comply with the rule. There are three options when the data is bad and you cannot enable the constraint:

- Delete the data you inserted.
- Enable the constraint using an exceptions table—this is discussed in Section 3.9, "Data Loads."
- Enable the constraint using the NOVALIDATE option. This syntax is:

```
ALTER TABLE table_name ENABLE CONSTRAINT
constraint_name NOVALIDATE;
```

The NOVALIDATE option enables the constraint but for future transactions. The data present in the table does not have to comply with the constraint. This can have some serious consequences. For example, many components of an application rely on business rules. A text box on an HTML form may be populated with a query that uses a primary key in the WHERE clause—this is a situation where the program assumes that one row is returned. This code will crash if it runs and multiple rows are returned.

The NOVALIDATE option poses no threat if the nature of the task is to study and analyze historical data. You can leave the old, noncompliant data in the table, enable constraints, and proceed with loading some new data that you wish to conform to your business rules. But the NOVALIDATE option on an existing production system can break many applications.

When you DISABLE a primary key or unique constraint, the index is dropped. So, if the table is large, you may see some time delay when you enable the constraint. This is the time it would take to rebuild the index. The same holds if you DROP the constraint; the index is dropped.

You use a CASCADE option when you DISABLE or DROP a primary key of unique constraint that is referenced by a foreign key. This syntax is:

```
ALTER TABLE table_name
DISABLE CONSTRAINT constraint_name CASCADE;

ALTER TABLE table_name
DROP CONSTRAINT constraint_name CASCADE;
```

You cannot accidentally corrupt the enforcement of referential integrity. If you don't realize there is a foreign key constraint and skip the cascade option, the ALTER TABLE fails with the Oracle error:

```
ORA-02272: this unique/primary key is referenced by some
foreign key
```

3.8 Exception Handling

The PL/SQL built-in exception DUP_VAL_ON_INDEX is raised whenever a SQL statement violates a primary key constraint. This is actually a PL/SQL built-in exception that is raised because you attempted to duplicate a column value for which there is a unique index—that index being the index generated on your behalf when you created the primary key. At the time such an error occurs, the Oracle error will be ORA-00001 and you can capture that error code with the PL/SQL SQLCODE built-in function. The following stored procedure implements the same functionality as the aforementioned Java procedure.

```
create or replace procedure INSERT_STUDENT
(
    v_student_name   students.student_name%TYPE,
    v_college_major  students.college_major%TYPE,
    v_status         students.status%TYPE
)
IS
BEGIN
    INSERT INTO students (student_id, student_name,
        college_major, status)
    VALUES (students_pk_seq.nextval,
            v_student_name,
            v_college_major,
            v_status);
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        dbms_output.put_line('We have a duplicate insert');
        dbms_output.put_line('SQLERRM:'||SQLERRM);
        dbms_output.put_line('SQLCODE:'||SQLCODE);
END insert_student;
```

Should the aforementioned procedure fail due to a duplicate primary key value, the output will be the following:

```
We have a duplicate insert
SQLERRM:ORA-00001: unique constraint (SCOTT.PK_STUDENTS)
violated
SQLCODE:-1
```

The SQLCODE is an Oracle predefined function, that only has scope within an exception handler. The value of SQLCODE is not always the same as the Oracle ORA error number. In PL/SQL, your best approach to capture this specific constraint violation is to have an exception handler on the DUP_VAL_ON_INDEX exception. If your code ever enters that program scope,

then you are sure you committed to either a primary key or unique constraint violation.

When developing applications with other languages, you need to look at the drivers. The Java code we see in Section 3.1.5, "Sequences in Code," uses the getErrorCode() method, which does not return a minus 1 but the five-digit number—for a primary key constraint violation, a 1.

We have discussed error handling from an end user perspective; that is, capture primary constraint violations, no matter how rare they might be, and respond to the user with a meaningful message, still leaving the user connected to the application.

3.9 Data Loads

Databases frequently have multiple information providers as shown in Figure 3–9.

OLTP providers usually perform single-row inserts or updates—these transactions are usually just a few rows. Data can also be loaded in from other systems. These are batch data loads and occur when there is less OLTP activity.

Data loads typically fall into two categories. One type is a schema initialization load. This process brings data into the database for the first time and coincides with the application development. The load may be from a legacy system that is being converted to Oracle. These loads require data "scrubbing" (e.g., the legacy may require string conversions to load time/day fields into an Oracle DATE type). Constraints and indexes can be built after the data is verified and loaded.

Other batch loads occur on a periodic base. A load can initiate from a user, an operating system-scheduled job, or possibly a PL/SQL procedure scheduled through the Oracle DBMS_JOB queue.

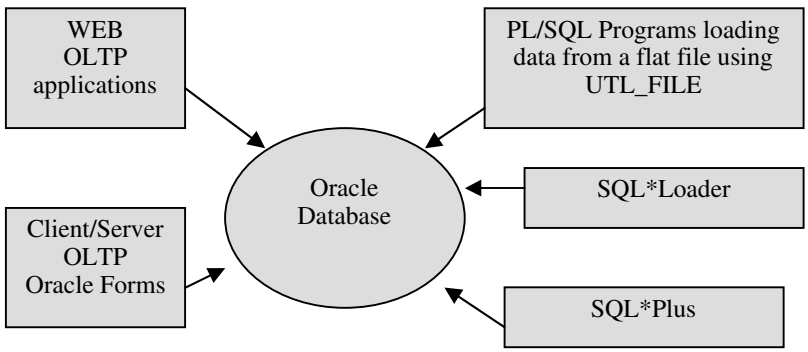

Figure 3–9 *Database Information Providers.*

SQL*Loader is an Oracle utility for loading fixed-format or delimited fields from an ASCII file into a database. It has a conventional and direct load option. The default option is conventional.

The direct load method disables constraints before the data load and enables them afterward. This incurs some overhead. For large amounts of data, the direct method is much faster than the conventional method. Postload processing includes not just the enabling of constraints, but the rebuilding of indexes for primary key and unique constraints.

If a direct load contains duplicates, the post process of enabling constraints and rebuilding of indexes fails. For a duplicate primary key or unique constraint, the failed state leaves the index in a "direct load" state.

Log messages in the SQL*Loader file will highlight this type of failure with the following:

```
The following index(es) on table STUDENTS were processed:
Index PK_STUDENTS was left in Direct Load State due to
ORA-0145 cannot CREATE UNIQUE INDEX; duplicate keys found
```

Following a direct load, you should check the SQL*Loader log file but also check the status of your indexes. A simple query for troubleshooting is the following:

```
SELECT index_name, table_name
FROM USER_INDEXES WHERE STATUS = 'DIRECT LOAD';
```

If you have bogus data following a direct load, you need to remove all duplicates before you can enable constraints and rebuild the indexes.

For a conventional SQL*Loader path, duplicate records are written to the SQL*Loader "bad" file with corresponding messages in the SQL*Loader "log" file. If no errors occur, then there is no "bad" file. SQL*Loader is a callable program and can be invoked in a client/server environment where the end user takes an action to load a file that is stored on the server. The mere existence of a bad file, following the load, will indicate errors during the load.

You can use SQL*Loader as a callable program to implement daily loads using a conventional path. You can use this utility to load large files with millions of rows into a database with excellent results.

Each SQL*Loader option (conventional and direct load) provides a mechanism to trap and resolve records that conflict with your primary key or any other constraint; however, direct load scenarios can be more time consuming.

Alternatives to SQL*Loader are SQL*Plus scripts and PL/SQL. You can load the data with constraints on and capture failed records through exception handling. Bad records can be written to a file using the UTL_FILE package. Bad records can also be written to a temporary table that has no constraints.

You also have the option to disable constraints, load data into a table, and then enable the constraint. If the data is bad you cannot enable the

constraint. To resolve bad records, start with an EXCEPTIONS table. The exceptions table can have any name, but must have the following columns.

```
CREATE TABLE EXCEPTIONS
 (row_id     ROWID,
  owner      VARCHR2(30),
  table_name VARCHAR2(30),
  constraint VARCHAR2(30));
```

The SQL for this exceptions table is found in the ORACLE_HOME/RDBMS/ADMIN directory in the file utlecpt.sql. The RDBMS/ADMIN directory, under ORACLE_HOME, is the standard repository for many scripts including the SQL scripts to build the data dictionary catalog, scripts to compile the SYS packages, and scripts like the exceptions table.

We use the exceptions table to capture rows that violate a constraint. This capturing is done as we attempt to enable our constraint. The following TEMP table is created with a primary key.

```
CREATE TABLE TEMP
(id   VARCHAR2(5) CONSTRAINT PK_TEMP PRIMARY KEY,
 no NUMBER);
```

Insert some good data:

```
INSERT INTO temp VALUES ('AAA', 1);
INSERT INTO temp VALUES ('BBB', 2);
```

The following disables the constraint. This is done here prior to inserting new data.

```
ALTER TABLE temp DISABLE CONSTRAINT PK_TEMP;
```

Now we insert some data; in this example, this is one row that we know to be duplicate row.

```
INSERT INTO temp VALUES ('AAA', 3);
```

The following shows the error when we enable the constraint with SQL*Plus.

```
SQL> ALTER TABLE temp ENABLE CONSTRAINT pk_temp;

ORA-00001 cannot enable constraint. Unique constraint
pk_temp violated.

SQL>
```

What if we had started with a million rows in TEMP and loaded another million. The task of identifying the offending rows can be tedious. Use an exceptions table when enabling constraints. The exceptions table captures the ROW ID of all offending rows.

```
ALTER TABLE temp ENABLE CONSTRAINT pk_temp EXCEPTIONS INTO
exceptions;
```

The constraints are still off. All records are in TEMP, but you can identify the bad records.

```
SELECT id, no
FROM   temp, exceptions
WHERE  exceptions.constraint='PK_TEMP'
AND    temp.rowid=exceptions.row_id;
```

This works for all types of constraint violations. You may not be able to enable constraints after a load, but you can capture the ROWID and constraint type through an exceptions table.