CHAPTER 7

Data Definition, Part II

Chapter 3 introduced just enough data definition (DDL) syntax to enable you to create the seven case tables for this book, using simple `CREATE TABLE` commands without any constraint specifications. This second DDL chapter goes into more detail about some data definition aspects, although it is still not intended as a complete reference on the topic. (Discussion of the `CREATE TABLE` command alone covers more than 100 pages in the Oracle Database 10*g* documentation.)

The first two sections revisit the `CREATE TABLE` command and the datatypes supported by Oracle Database 10*g*. Section 7.3 introduces the `ALTER TABLE` command, which allows you to change the structure of an existing table (such as to add columns or change datatypes), and the `RENAME` command, which allows you to rename a table or view. You will learn how to define and handle constraints in Section 7.4.

Section 7.5 covers indexes. The main purpose of indexes is to improve performance (response time) by providing more efficient access paths to table data. Thus, Section 7.6 provides a brief introduction to performance, mainly in the context of checking if the optimizer is using your indexes.

The most efficient method to generate sequence numbers (for example, for order numbers) in an Oracle environment is by using sequences, which are introduced in Section 7.7.

We continue with synonyms, in Section 7.8. By creating synonyms you can work with abbreviations for table names, hide the schema name prefix of table names, or even hide the remote database where the table resides. Section 7.9 explains the `CURRENT_SCHEMA` session parameter.

Section 7.10 discusses the `DROP TABLE` command and the recycle bin, a concept introduced in Oracle Database 10*g*. By default, all dropped tables go to the recycle bin, allowing you to recover from human errors.

The next two sections cover some other SQL commands related to data definition: `TRUNCATE` and `COMMENT`. The final section contains some review exercises.

7.1 The CREATE TABLE Command

Chapter 3 introduced the `CREATE TABLE` command and showed a basic command syntax diagram. This section explores the `CREATE TABLE` command in a little more detail. Figure 7-1 shows a more (but still far from) complete syntax diagram.

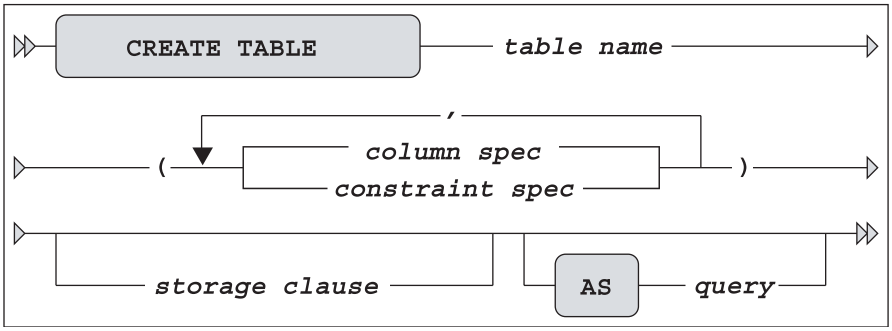

Figure 7-1. *CREATE TABLE command syntax diagram*

Figure 7-1 shows that the `CREATE TABLE` command supports two component types: *column* specifications and *constraint* specifications.

You can provide an optional `STORAGE` clause, with various physical storage specifications for the table you are creating. This is an important means to optimize and spread the physical storage of your data on disk. For more information about the `STORAGE` clause and handling physical storage, see *Oracle SQL Reference*.

According to the syntax diagram in Figure 7-1, you can also create new tables based on a subquery with the `AS` clause. The `CREATE TABLE ... AS SELECT ...` command (also known as CTAS) is comparable to one of the possibilities of the `INSERT` command shown in Figure 6-1 (in Chapter 6), where you insert rows into an existing table using a subquery. The only difference is that with CTAS, you create *and* populate the table in a single SQL command. In this case, you can omit the column specifications between the parentheses. If you want to use column specifications anyway, you are not allowed to specify datatypes. In CTAS commands, the new table always inherits the datatypes from the results of the subquery.

The syntax for column specifications in a `CREATE TABLE` command is detailed in Figure 7-2.

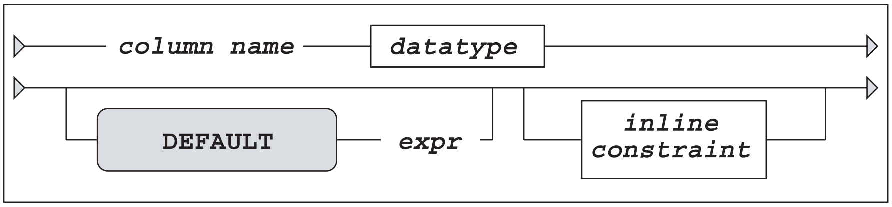

Figure 7-2. *CREATE TABLE column specification syntax*

Figure 7-2 shows that you can specify constraints in two ways:

- As independent (*out-of-line)* components of the `CREATE TABLE` command (see Figure 7-1)
- As *inline constraints* inside a column specification (see Figure 7-2)

We will discuss both types of constraints in Section 7.4.

You can use the `DEFAULT` option to specify a value (or an expression) to be used for `INSERT` commands that don't contain an explicit value for the corresponding column.

7.2 More on Datatypes

Datatypes were introduced in Chapter 3. Table 7-1 provides a more complete overview of the most important Oracle datatypes.

Table 7-1. *Important Oracle Datatypes*

Datatype	Description
`CHAR[(n)]`	Character string with fixed length *n* (default 1)
`VARCHAR2(n)`	Variable-length string; maximum *n* characters
`DATE`	Date (between 4712 BC and 9999 AD)
`TIMESTAMP`	Timestamp, with or without time zone information
`INTERVAL`	Date/time interval
`BLOB`	Unstructured binary data (Binary Large Object)
`CLOB`	Large text (Character Large Object)
`RAW(n)`	Binary data; maximum *n* bytes
`NUMBER`	Integer; maximum precision 38 digits
`NUMBER(n)`	Integer; maximum *n* digits
`NUMBER(n,m)`	Total of *n* digits; maximum *m* digits right of the decimal point
`BINARY_FLOAT`	32-bit floating-point number
`BINARY_DOUBLE`	64-bit floating-point number

■**Note** If you insert values into a `NUMBER(n,m)` column and you exceed precision *n*, you get an error message. If you exceed scale *m*, the Oracle DBMS rounds the value.

The Oracle DBMS supports many datatype synonyms for portability with other DBMS implementations and for compliance with the ANSI/ISO standard. For example, `CHARACTER` is identical to `CHAR`; `DECIMAL(n,m)` is identical to `NUMBER(n,m)`; and `NUMBER` even has multiple synonyms, such as `INTEGER`, `REAL`, and `SMALLINT`.

Each Oracle datatype has its own precision or length limits, as shown in Table 7-2.

Table 7-2. *Oracle Datatype Limits*

Datatype	Limit
`NUMBER`	38 digits
`CHAR`	2000
`VARCHAR2`	4000
`RAW`	2000 bytes
`BLOB`	(4GB – 1) × (database block size)
`CLOB`	(4GB – 1) × (database block size)

Character Datatypes

Since Oracle7 (released more than ten years ago), `VARCHAR` and `VARCHAR2` have exactly the same meaning. However, Oracle recommends using the `VARCHAR2` datatype, because a future Oracle release might treat those two datatypes differently.

If you go a little further back in time (Oracle version 6 and earlier), the datatypes `CHAR` and `VARCHAR` were synonyms, both representing *variable*-length character strings; the Oracle DBMS didn't support fixed-length strings. The change in behavior in Oracle7 caused a lot of problems, although it was announced in advance. This is one of the reasons why we now have `VARCHAR` and `VARCHAR2`.

You may have noticed that Table 7-2 shows 2000 and 4000 for the `CHAR` and `VARCHAR2` datatype limits, respectively. You might wonder in which unit these numbers are expressed. That depends on the value of the `NLS_LENGTH_SEMANTICS` parameter. The default for the Oracle DBMS is to use `BYTE` length semantics. If you want to make your SQL code independent of this parameter, you can override its value by using explicit `BYTE` and `CHAR` suffixes in your datatype specifications. Here are a couple examples:

- `CHAR(42 BYTE)`: Fixed string, 42 bytes
- `VARCHAR2(2000 CHAR)`: Variable string, maximum of 2000 characters

Comparison Semantics

If `VARCHAR2` and `VARCHAR` do diverge in the future, the `VARCHAR2` datatype will be guaranteed to be backward-compatible. The eventual difference between these two datatypes could be the treatment of comparisons involving strings of different lengths, or maybe the interpretation of empty strings as null values. There are two different semantics to compare strings of different lengths: padded comparison (padding with spaces) and nonpadded comparison.

If you compare two strings, character by character, and all of the characters are identical until the point where the shortest string is processed, nonpadded comparison semantics automatically "declares" the longest string as being greater than the shorter string. On the other hand, padded comparison semantics extends the shortest string with spaces until the length of the longest string, and continues comparing characters. This means that trailing spaces in strings don't influence padded comparison results. Here are examples of the comparison types:

- Padded comparison: `'RAID5' = 'RAID5   '`
- Nonpadded comparison: `' RAID5' < ' RAID5   '`

By using the `VARCHAR2` datatype instead of the `VARCHAR` datatype, especially in all your SQL script files, you are guaranteed to get *nonpadded* comparison semantics, regardless of the development and implementation of the `VARCHAR` datatype in any future release of the Oracle DBMS.

Column Data Interpretation

There is an important difference between the `RAW` and `VARCHAR2` datatypes. `RAW` column data (like `BLOB` data) is never interpreted by the DBMS in any way. For example, `VARCHAR2` column data is converted automatically during transport from an ASCII to an EBCDIC environment. You typically use the RAW and BLOB datatypes for columns containing binary data, such as scanned documents, sound tracks, and movie fragments.

Numbers Revisited

Before we move on to the ALTER TABLE command in the next section, let's briefly revisit numbers. The Oracle DBMS has always stored NUMBER values in a proprietary internal format, to maintain maximum *portability* to the impressive list of different platforms (operating systems) that it supports. The NUMBER datatype is still the best choice for most columns containing numeric data. However, the internal storage of this datatype implies some processing overhead, especially when you are performing many nontrivial numerical computations in your SQL statements.

Since Oracle Database 10*g* you can also store *floating-point* numbers in your table columns. Floating-point numbers don't offer the same precision as NUMBER values, but they may result in better response times for numerical computations. You can choose between two floating-point datatypes:

- BINARY_FLOAT: 32-bit, single precision
- BINARY_DOUBLE: 64-bit, double precision

You can also specify floating-point constants (literals) in your SQL statements with a suffix f (single precision) or d (double precision), as shown in Listing 7-1.

Listing 7-1. *Floating-Point Literals*

```
SQL> select 5.1d, 42f from dual;

      5.1D        42F
---------- ----------
   5.1E+000   4.2E+001

SQL>
```

We won't use these two floating-point datatypes in this book. See *Oracle SQL Reference* for more details.

7.3 The ALTER TABLE and RENAME Commands

Sometimes, it is necessary to change the structure of existing tables. For example, you may find that the maximum width of a certain column is defined too low, you might want to add an extra column to an existing table, or you may need to modify a constraint. In these situations, you can use the ALTER TABLE command. Figure 7-3 shows the syntax diagram for this command.

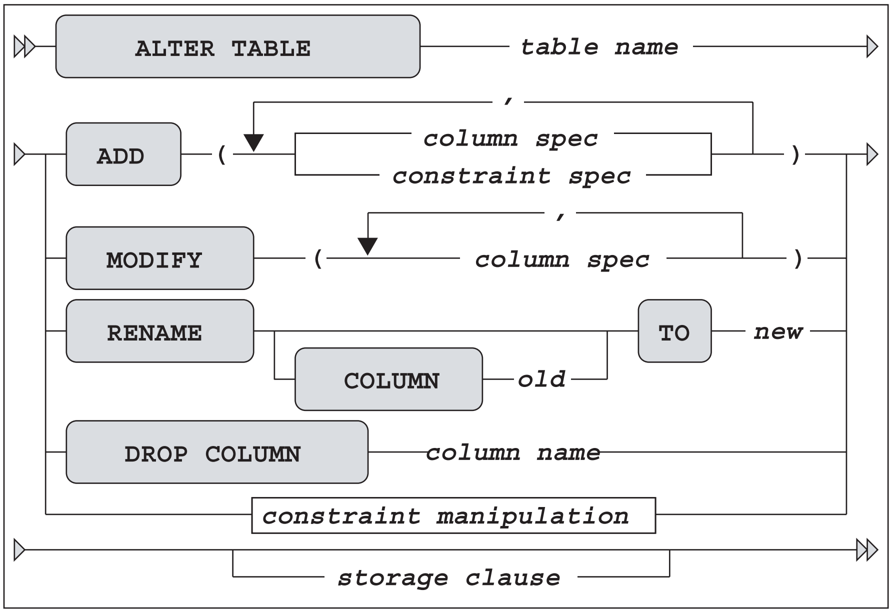

Figure 7-3. *ALTER TABLE command syntax diagram*

■**Note** The `ALTER TABLE` command is much more complicated and extended than Figure 7-3 suggests. See *Oracle SQL Reference* for more details.

You can add columns or constraint definitions to an existing table with the `ADD` option. The `MODIFY` option allows you to change definitions of existing columns. For example, you can widen a column, allow null values with `NULL`, or prohibit null values with `NOT NULL`.

You can drop columns from tables with the `DROP COLUMN` option. You can also set columns to "unused" with the `ALTER TABLE ... SET UNUSED` command, and physically remove them from the database later with the `ALTER TABLE ... DROP UNUSED COLUMNS` command. This may be useful when you want to drop multiple columns in a single scan (accessing the rows only once). The `RENAME COLUMN` option allows you to change the name of a column.

■**Caution** You should be careful with the "destructive" `DROP COLUMN` option. Some database applications may depend on the existence of the column you are dropping.

With the *`constraint manipulation`* option, you can remove, enable, or disable constraints. Figure 7-4 shows the syntax details of this `ALTER TABLE` command option. For more details about constraint handling, see the next section.

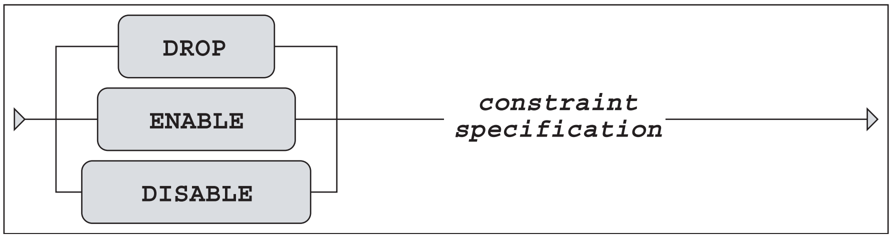

Figure 7-4. *ALTER TABLE constraint manipulation syntax*

Just like the `CREATE TABLE` command, the `ALTER TABLE` command also allows you to influence various physical table storage attributes.

In general, you can apply any structure change to existing tables, even when they contain rows. However, there are some exceptions. For example, for obvious reasons you cannot add a `NOT NULL` column to a nonempty table, unless you immediately specify a `DEFAULT` value in the same `ALTER TABLE` command. Listing 7-2 shows an example.

Listing 7-2. *ALTER TABLE Command Examples*

```
SQL> alter table registrations
  2  add  (entered_by number(4) default 7839 not null);

Table altered.

SQL> alter table registrations
  2  drop  column entered_by;

Table altered.

SQL>
```

Note The `ALTER TABLE` statement is probably the best illustration of the power of the relational model. Think about this: you can change a table definition while the table contains data and applications are running.

The `RENAME` command is rather straightforward. It allows you to change the name of a table or view (views are discussed in Chapter 10). Figure 7-5 shows the syntax diagram for the `RENAME` command.

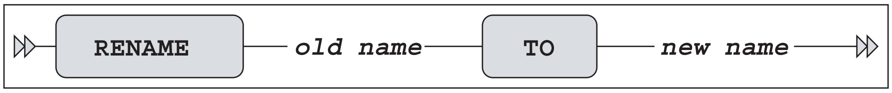

Figure 7-5. *RENAME command syntax diagram*

7.4 Constraints

As you saw in the previous sections, you can specify constraint definitions in the CREATE TABLE and ALTER TABLE commands. As noted earlier in the description of the CREATE TABLE command, you can treat constraints as independent table components (for example, at the end of your CREATE TABLE command after all column definitions) or as part of a column definition. A common terminology to distinguish these two ways to specify constraints is *out-of-line* versus *inline* constraints.

For each constraint definition, you can optionally specify a constraint name. It is highly recommended that you do so for all your constraint definitions. If you don't specify a constraint name yourself, the Oracle DBMS generates a far from informative name for you: SYS_C*nnnnn*, where *nnnnn* is an arbitrary sequence number. Once constraints are created, you need their names to manipulate (enable, disable, or drop) them. Moreover, constraint names show up in constraint violation error messages. Therefore, well-chosen constraint names make error messages more informative. See Listing 7-3 later in this section for an example, showing a foreign key constraint violation.

Out-of-Line Constraints

Figure 7-6 shows the syntax details for out-of-line constraints. This syntax is slightly different from the inline constraint syntax.

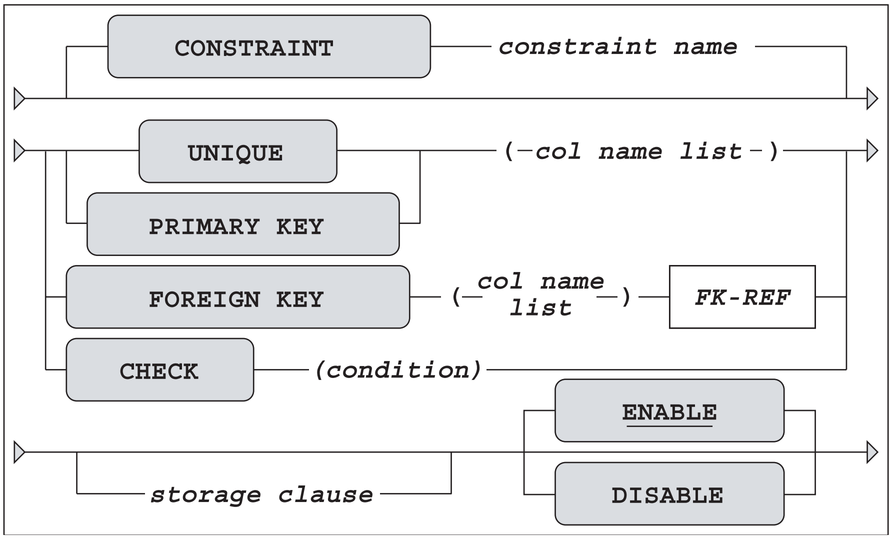

Figure 7-6. *Out-of-line constraint syntax diagram*

In the syntax diagram, *col name list* refers to a comma-separated list of one or more column names. The type of constraint can be UNIQUE, PRIMARY KEY, FOREIGN KEY, and CHECK. By default, constraints become active immediately, unless you specify the DISABLE option; in other words, the default option is ENABLE.

The four types of constraints work as follows:

- UNIQUE allows you to prevent duplicate values in a column or a column combination.
- PRIMARY KEY and FOREIGN KEY allow you to implement *entity integrity* and *referential integrity*. See Chapter 1 for a detailed discussion of these concepts.
- CHECK allows you to specify any arbitrary *condition* as a constraint.

Figure 7-7 shows the syntax details of a foreign key constraint reference (*FK-REF* in Figure 7-6).

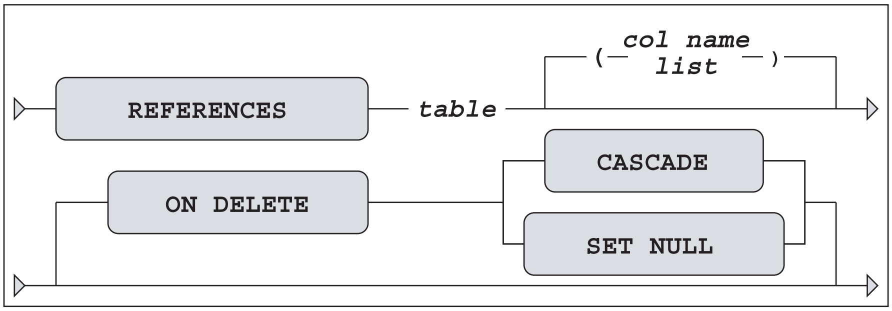

Figure 7-7. *Foreign key reference syntax diagram*

You can omit the comma-separated list of column names (*col name list* in Figure 7-7) in the foreign key reference. In that case, the foreign key constraint automatically refers to the primary key of the referenced table.

■ **Tip** In general, it is considered good practice to have foreign keys always refer to primary keys, although foreign keys may also reference unique keys.

To understand the **ON DELETE** option of the foreign key reference, consider the example of a foreign key constraint violation shown in Listing 7-3. Normally, it is impossible to remove parent (master) rows if the database still contains child (detail) rows. In Listing 7-3, we try to remove the XML course while the database still apparently contains XML course offerings.

Listing 7-3. *Example of a Foreign Key Constraint Violation*

```
SQL> delete from courses
  2  where  code = 'XML';

delete from courses
*
ERROR at line 1:
ORA-02292: integrity constraint (BOOK.O_COURSE_FK) violated -
           child record found

SQL>
```

■Note Listing 7-10 shows the definition of the `O_COURSE_FK` constraint.

The `ON DELETE CASCADE` option (see Figure 7-7) changes the behavior in such situations. The master/detail problems are solved by a cascading effect, in which, apart from the parent row, all child rows are implicitly deleted, too. The `ON DELETE SET NULL` option solves the same problem in a different way: the child rows are updated, rather than deleted. This approach is applicable only if the foreign key columns involved may contain null values, of course.

Inline Constraints

The *inline* constraint syntax is shown in Figure 7-8. There are some subtle differences from the syntax for out-of-line constraints:

- You don't specify column names in inline constraints, because inline constraints always belong to the column definition in which they are embedded.
- The foreign key constraint reference (*FK-REF*) is the same for both constraint types (see Figure 7-7), but you don't specify the keywords `FOREIGN KEY` for an inline constraint—`REFERENCES` is enough.
- In the context of inline constraints, a `NOT NULL` constraint is allowed. In out-of-line constraints, this is impossible, unless you rewrite it as a `CHECK` constraint.

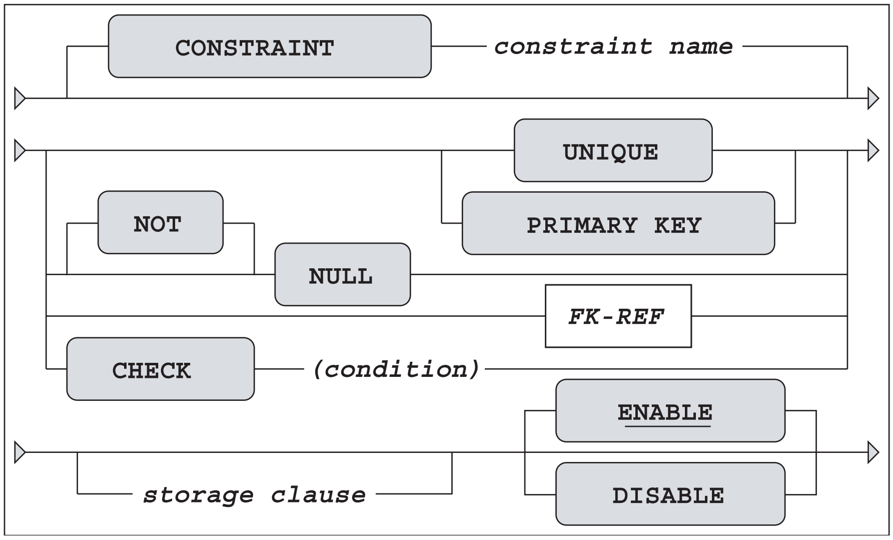

Figure 7-8. *Inline constraint syntax diagram*

Constraint Definitions in the Data Dictionary

Constraint definitions are stored in the data dictionary. The two most important views are USER_CONSTRAINTS and USER_CONS_COLUMNS. Listing 7-4 shows how you can produce an overview of all referential integrity constraints for the current user.

Listing 7-4. *Foreign Key Constraints in the Data Dictionary*

```
SQL> select table_name
  2  ,      constraint_name
  3  ,      status
  4  ,      r_constraint_name as references
  5  from   user_constraints
  6  where  constraint_type = 'R';

TABLE_NAME           CONSTRAINT_NAME      STATUS   REFERENCES
-------------------- -------------------- -------- ----------
EMPLOYEES            E_MGR_FK             ENABLED  E_PK
DEPARTMENTS          D_MGR_FK             ENABLED  E_PK
EMPLOYEES            E_DEPT_FK            ENABLED  D_PK
OFFERINGS            O_TRAIN_FK           ENABLED  E_PK
OFFERINGS            O_COURSE_FK          ENABLED  C_PK
REGISTRATIONS        R_OFF_FK             ENABLED  O_PK
REGISTRATIONS        R_ATT_FK             ENABLED  E_PK
HISTORY              H_DEPT_FK            ENABLED  D_PK
HISTORY              H_EMPNO_FK           ENABLED  E_PK

SQL>
```

Tools like Oracle Forms and Oracle Designer can use constraint definitions from the data dictionary; for example, to generate code for constraint checking in database applications.

Last, but not least, the Oracle optimizer uses knowledge about constraint information from the data dictionary to decide about efficient execution plans for SQL statements. To reiterate what we discussed in Chapter 1, constraints are very important, and they *must* be defined in the database.

Case Table Definitions with Constraints

Listings 7-5 through 7-12 show the CREATE TABLE commands for the seven case tables of this book. The constraints in these CREATE TABLE commands are meant to be self-explanatory, showing various examples of PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK, and NOT NULL constraints.

■**Note** For more details about the seven case tables, refer to Appendix C of this book.

Listing 7-5. *The EMPLOYEES Table*

```
create table employees
( empno      NUMBER(4)    constraint E_PK        primary key
                          constraint E_EMPNO_CHK check (empno > 7000)
, ename      VARCHAR2(8)  constraint E_NAME_NN   not null
, init       VARCHAR2(5)  constraint E_INIT_NN   not null
, job        VARCHAR2(8)
, mgr        NUMBER(4)    constraint E_MGR_FK    references employees
, bdate      DATE         constraint E_BDAT_NN   not null
, msal       NUMBER(6,2)  constraint E_MSAL_NN   not null
, comm       NUMBER(6,2)
, deptno     NUMBER(2)    default 10
,                         constraint E_SALES_CHK check
                                                 (decode(job,'SALESREP',0,1)
                                                  + nvl2(comm,          1,0) = 1)
) ;
```

Listing 7-6. *The DEPARTMENTS Table*

```
create table departments
( deptno NUMBER(2)     constraint D_PK          primary key
                       constraint D_DEPTNO_CHK  check (mod(deptno,10) = 0)
, dname  VARCHAR2(10)  constraint D_DNAME_NN    not null
                       constraint D_DNAME_UN    unique
                       constraint D_DNAME_CHK   check (dname = upper(dname))
, location VARCHAR2(8) constraint D_LOC_NN      not null
                       constraint D_LOC_CHK     check (location = upper(location))
, mgr    NUMBER(4)     constraint D_MGR_FK      references employees
) ;
```

Listing 7-7. *Adding a Foreign Key Constraint*

```
alter table employees add
(constraint E_DEPT_FK foreign key (deptno) references departments);
```

Listing 7-8. *The SALGRADES Table*

```
create table salgrades
( grade      NUMBER(2)   constraint S_PK        primary key
, lowerlimit NUMBER(6,2) constraint S_LOWER_NN  not null
                         constraint S_LOWER_CHK check (lowerlimit >= 0)
, upperlimit NUMBER(6,2) constraint S_UPPER_NN  not null
, bonus      NUMBER(6,2) constraint S_BONUS_NN  not null
,                        constraint S_LO_UP_CHK check
                                                (lowerlimit <= upperlimit)
) ;
```

Listing 7-9. *The COURSES Table*

```
create table courses
( code        VARCHAR2(6)  constraint C_PK       primary key
, description VARCHAR2(30) constraint C_DESC_NN  not null
, category    CHAR(3)      constraint C_CAT_NN   not null
, duration    NUMBER(2)    constraint C_DUR_NN   not null
,                          constraint C_CODE_CHK check
                                                 (code = upper(code))
,                          constraint C_CAT_CHK  check
                                                 (category in ('GEN','BLD','DSG'))
) ;
```

Listing 7-10. *The OFFERINGS Table*

```
create table offerings
( course     VARCHAR2(6)  constraint O_COURSE_NN not null
                          constraint O_COURSE_FK references courses
, begindate  DATE         constraint O_BEGIN_NN  not null
, trainer    NUMBER(4)    constraint O_TRAIN_FK  references employees
, location   VARCHAR2(8)
,                         constraint O_PK        primary key
                                                 (course,begindate)
) ;
```

Listing 7-11. *The REGISTRATIONS Table*

```
create table registrations
( attendee    NUMBER(4)   constraint R_ATT_NN    not null
                          constraint R_ATT_FK    references employees
, course      VARCHAR2(6) constraint R_COURSE_NN not null
, begindate   DATE        constraint R_BEGIN_NN  not null
, evaluation  NUMBER(1)   constraint R_EVAL_CHK  check (evaluation in (1,2,3,4,5))
,                         constraint R_PK        primary key
                                                 (attendee,course,begindate)
,                         constraint R_OFF_FK    foreign key (course,begindate)
                                                 references offerings
) ;
```

Listing 7-12. *The HISTORY Table*

```
create table history
( empno      NUMBER(4)    constraint H_EMPNO_NN  not null
                          constraint H_EMPNO_FK  references employees
                                                 on delete cascade
, beginyear  NUMBER(4)    constraint H_BYEAR_NN  not null
, begindate  DATE         constraint H_BDATE_NN  not null
, enddate    DATE
, deptno     NUMBER(2)    constraint H_DEPT_NN   not null
                          constraint H_DEPT_FK   references departments
, msal       NUMBER(6,2)  constraint H_MSAL_NN   not null
, comments   VARCHAR2(60)
,                         constraint H_PK        primary key (empno,begindate)
,                         constraint H_BEG_END   check (begindate < enddate)
) ;
```

A Solution for Foreign Key References: CREATE SCHEMA

While we are on the topic of creating multiple tables, Oracle SQL also supports the ANSI/ISO standard `CREATE SCHEMA` command. This command allows you to create a complete schema (consisting of tables, views, and grants) with a single DDL command/transaction. One advantage of the `CREATE SCHEMA` command is that it succeeds or fails as an atomic transaction. It also solves the problem of two tables having foreign key references to each other (see Listings 7-5, 7-6, and 7-7), where you normally need at least one `ALTER TABLE` command, because foreign keys can reference only existing tables.

Listing 7-13 shows how you could have created the case tables with the `CREATE SCHEMA` command.

Listing 7-13. *The CREATE SCHEMA Command*

```
SQL> create schema authorization BOOK
  2        create table employees     (...)
  3        create table departments   (...)
  4        create table salgrades     (...)
  5        create table courses       (...)
  6        create table offerings     (...)
  7        create table registrations (...)
  8        create table history       (...)
  9        create view ... as select ... from ...
 10        grant select on ... to public;
```

■**Note** The name of this command (as implemented by Oracle) is confusing, because it does not actually create a schema. Oracle schemas are created with the `CREATE USER` command. The command succeeds only if the schema name is the same as your Oracle database username.

You can specify the `CREATE SCHEMA` command components in any order. Within each component definition, you can refer to other (earlier or later) schema components.

Deferrable Constraints

The Oracle DBMS also supports *deferrable constraints*, allowing you to specify *when* you want the constraints to be checked. These are the two possibilities:

- `IMMEDIATE` checks at the statement level.
- `DEFERRED` checks at the end of the transaction.

Before you can use this distinction, you must first allow a constraint to be deferrable. The default option for all constraints that you create is `NOT DEFERRABLE`. If you want your constraints to be deferrable, add the `DEFERRABLE` option in the constraint definition, as shown in Figure 7-9, just before the *`storage clause`* specification (see Figures 7-6 and 7-8).

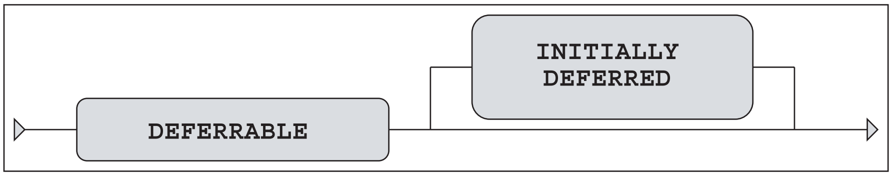

Figure 7-9. *DEFERRABLE option for constraint definitions*

If you allow constraints to be deferrable using the DEFERRABLE option, they still have a default behavior of INITIALLY IMMEDIATE. The INITIALLY option allows you to specify the desired default constraint checking behavior, using IMMEDIATE or DEFERRED.

You can dynamically change or override the default behavior of deferrable constraints at the transaction level with the SET CONSTRAINTS command, as shown in Figure 7-10.

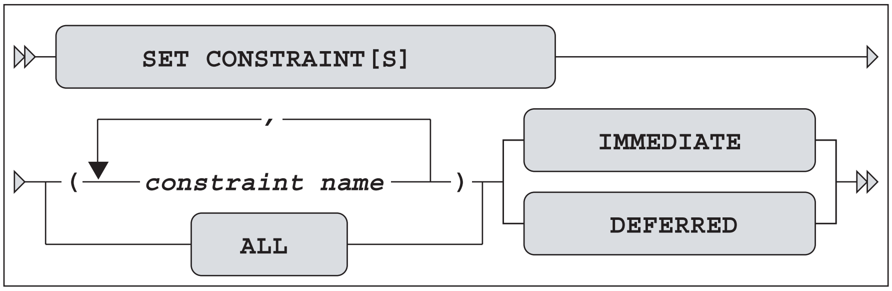

Figure 7-10. *SET CONSTRAINTS command syntax diagram*

At first sight, the complexity of all this constraint-checking syntax may look overwhelming. The following summary may help clarify how it works:

- By default, the Oracle DBMS *always* uses immediate constraint checking.
- You must explicitly allow a constraint to be deferrable. By default, constraints are *not* deferrable.
- If constraints are deferrable, you can choose how they should be checked *by default*: immediate or deferred.
- If constraints are deferrable, you can influence their behavior with the SET CONSTRAINTS command.

7.5 Indexes

In general, rows within a regular table are unordered. Although the Oracle DBMS offers many different ways to physically organize tables on disk (heap tables, index clusters, hash clusters, index-organized tables, and sorted hash clusters), you should never expect the rows to be physically stored in a certain order. Even if a particular order exists today, there is no guarantee that it will be the same tomorrow. This is a fundamental property of relational databases (see Ted Codd's rule 8 in Chapter 1 about physical data independence).

Suppose the EMPLOYEES table contains 50,000 rows (instead of the 14 rows we have), and suppose you want to know which employees have a name starting with a *Q*. Normally, the Oracle DBMS can use only one method to produce the results for this query: by accessing all 50,000 rows (with a full table scan) and checking the name for each of those rows. This could take quite some time, and perhaps there would be no employees at all with such a name.

An *index* on employee names would be very useful in this situation. When you create an index, the Oracle DBMS creates, and starts to maintain, a separate database object containing a sorted list of column values (or column combination values) with row identifiers referring to the corresponding rows in the table. To further optimize access, indexes are internally organized in a tree structure. (See *Oracle Concepts* for more details on physical index structures.) If there were such an index on employee names, the optimizer could decide to abandon the full table scan approach and perform an index search instead. The index offers a very efficient access path to all names, returning all row identifiers of employees with a name starting with a *Q*. This probably would result in a huge performance improvement, because there are only a few database blocks to be visited to produce the query result.

For some of your other queries, indexes on department numbers or birth dates could be useful. You can create as many indexes per table as you like.

In summary, the performance of your SQL statements can often be improved significantly by creating indexes. Sometimes, it is obvious that an index will help, such as when your tables contain a lot of rows and your queries are very selective (only retrieving a few rows). On the other hand, though, you may find that your application benefits from an index on a single-row, single-column table.

Indexes may speed up queries, but the other side of the index picture is the maintenance overhead. Every additional index slows down data manipulation further, because every `INSERT/UPDATE/DELETE` statement against a table must immediately be processed against all corresponding indexes to keep the indexes synchronized with the table. Also, indexes occupy additional space in your database. This means that you should carefully consider which columns should be indexed and which ones should not be indexed.

These are some suggestions for index candidates:

- Foreign key columns
- Columns often used in `WHERE` clauses
- Columns often used in `ORDER BY` and `GROUP BY` clauses

Here, we'll look at the commands for index creation and management.

Index Creation

Figure 7-11 shows the (simplified) syntax of the `CREATE INDEX` command.

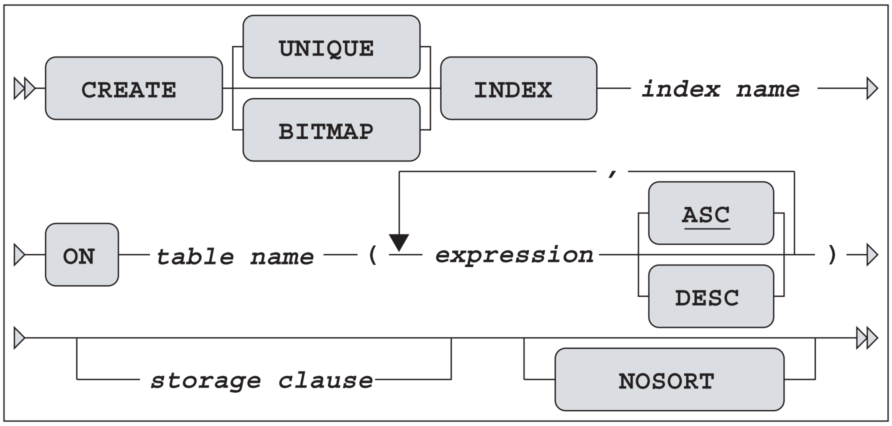

Figure 7-11. *CREATE INDEX command syntax diagram*

The *storage clause* allows you to influence various physical index storage attributes, such as the storage location and the space allocation behavior. See *Oracle SQL Reference* for more details. If the table rows happen to be inserted and stored in index order, you can specify the NOSORT option to speed up index creation. The Oracle DBMS will skip the sort phase (normally needed during index creation), but if the rows turn out to be in the wrong order, the CREATE INDEX command will fail with an error message.

Unique Indexes

Unique indexes serve two purposes: they provide additional access paths to improve response times (like nonunique indexes), and they also prevent duplicate values. You create unique indexes by specifying the UNIQUE option of the CREATE INDEX command (see Figure 7-11).

Note, however, that it is recommended to ensure uniqueness in your tables using the PRIMARY KEY and UNIQUE constraints, leaving it up to the Oracle DBMS to choose an appropriate physical implementation of those constraints.

Bitmap Indexes

Regular indexes work the best if the corresponding columns contain many different values, resulting in better selectivity. Unique indexes offer the best selectivity, because they contain only different values. This means that every equality search (... WHERE COL = ...) results in at most one row. At the other side of the spectrum, if a column contains only a few values (typical examples are gender, status, and yes/no columns), a regular index is not very useful, because the average selectivity of equality searches will be poor.

For such low-cardinality columns, the Oracle DBMS supports *bitmap indexes*. Bitmap indexes also outperform regular indexes if your WHERE clause is complicated, using many AND, OR, and NOT connectives. You create bitmap indexes by specifying the BITMAP option (see Figure 7-11).

■ **Caution** Indexes slow down data manipulation, and bitmap indexes are the most expensive index type in terms of maintenance. Don't create bitmap indexes on tables with a lot of DML activity.

Function-Based Indexes

As Figure 7-11 shows, you can specify an *expression* between the parentheses when defining the table columns to be indexed. That means that instead of simply specifying a single column or a comma-separated list of columns, you can choose to specify a more complicated expression in an index definition. Indexes containing such expressions are referred to as *function-based indexes*. See Listing 7-14 for an example, where we create an index on an expression for the yearly salary.

Listing 7-14. *Creating a Function-Based Index*

```
SQL> create index year_sal_idx
  2  on employees (12*msal + coalesce(comm,0));
Index created.

SQL>
```

The index we created in Listing 7-14 can provide an efficient access path for the Oracle DBMS to produce the result of the following query:

```
SQL> select * from employees where 12*msal+coalesce(comm,0) > 18000;
```

Function-based indexes can be used in combination with various NLS features to enable linguistic sorting and searching. See *Oracle SQL Reference* and *Oracle Globalization Support Guide* for more details. One of the exercises at the end of this chapter will ask you to create a function-based index for another specific purpose.

Index Management

Since indexes are maintained by the Oracle DBMS, each table change is immediately propagated to the indexes. In other words, indexes are always up-to-date. However, if your tables incur continuous and heavy DML activity, you might want to consider rebuilding your indexes. Of course, you could simply drop them and then re-create them. However, using the `ALTER INDEX ... REBUILD` or `ALTER INDEX ... COALESCE` command is more efficient. Figure 7-12 shows the (partial) syntax diagram for the `ALTER INDEX` command.

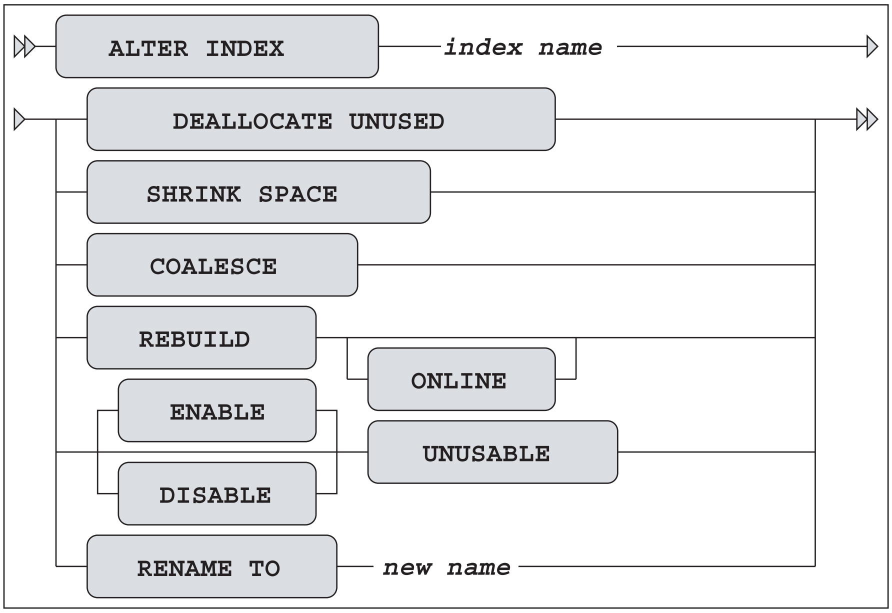

Figure 7-12. *ALTER INDEX command syntax diagram*

The various ALTER INDEX command options in Figure 7-12 (which is far from complete) show that this command belongs to the purview of database administrators, so we will not discuss them here.

■Note The ENABLE and DISABLE options of the ALTER INDEX command (see Figure 7-12) apply only to function-based indexes. If you set indexes to UNUSABLE, you must REBUILD (or DROP and CREATE) them before they can be used again.

You can remove indexes with the DROP INDEX command. Figure 7-13 shows the syntax diagram for DROP INDEX.

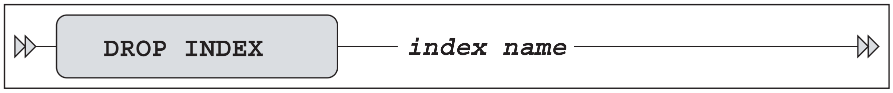

Figure 7-13. *DROP INDEX command syntax diagram*

Here is an example of removing an index:

```
SQL> drop index year_sal_idx;
Index dropped.

SQL>
```

■Tip In periods of heavy data-manipulation activity, without a lot of reporting (retrieval) activity, you may consider dropping indexes temporarily, and re-creating them later.

When you're working with indexes, keep in mind that although you can decide about index *existence* with the CREATE INDEX and DROP INDEX commands, the Oracle optimizer decides about index *usage*. The optimizer chooses the execution plan for each SQL statement. The next section explains how you can see if the optimizer is using your indexes.

7.6 Performance Monitoring with SQL*Plus AUTOTRACE

This is *not* a book about SQL performance tuning. However, in a chapter where we talk about creating indexes, it makes sense to at least show how you can see whether the indexes you create are actually used. What you need for that purpose is a way to see SQL execution plans.

Oracle provides many *diagnostic tools* (such as the SQL trace facility, TKPROF, and EXPLAIN PLAN) to help you with your performance-tuning efforts. However, discussion of these useful Oracle tools is not appropriate here; see *Oracle Performance Tuning Guide* for more details. Fortunately, SQL*Plus offers a limited but user-friendly alternative for those diagnostic tools: the AUTOTRACE facility.

If you want to use all of the options of the AUTOTRACE setting, you may need to prepare your Oracle environment:

- SQL*Plus assumes the existence of a PLAN_TABLE table to store execution plans. If necessary, you can create a local copy in your own schema with the utlxplan.sql script. Oracle Database 10*g* has a public synonym PLAN_TABLE, pointing to a global temporary table. Creating a local PLAN_TABLE is necessary only in earlier releases.
- You must have sufficient privileges for certain AUTOTRACE features. You need the PLUS_TRACE role, created by the plustrce.sql script. The plustrce.sql script must be executed from the SYSTEM database user account. If you don't have access to that privileged account, contact your local database administrator.

Both the utlxplan.sql and plustrce.sql scripts are shipped with the Oracle software. Listing 7-15 shows how you would run these scripts under Microsoft Windows. The question marks in these commands are interpreted as the directory where the Oracle software is installed on your machine.

Listing 7-15. *Preparing for SQL*Plus AUTOTRACE Usage*

```
SQL> connect system/manager
Connected.

SQL> @?\sqlplus\admin\plustrce
Role created.

SQL> grant plustrace to book;
Grant succeeded.

SQL> connect book/book
Connected.

SQL> @?\rdbms\admin\utlxplan
Table created.

SQL>
```

After you have prepared your environment, you can use AUTOTRACE. Figure 7-14 shows the syntax diagram for using AUTOTRACE.

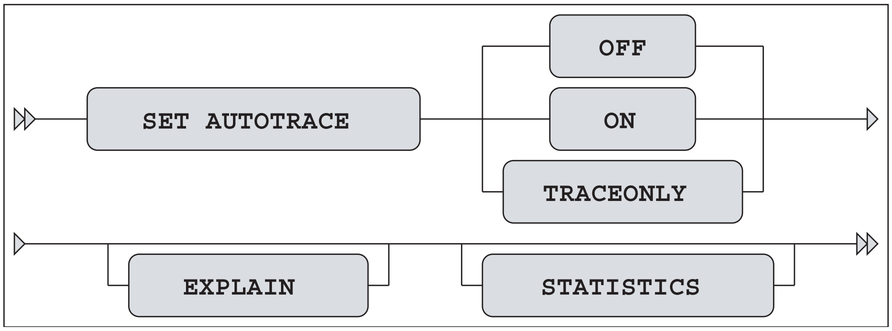

Figure 7-14. *SQL*Plus AUTOTRACE setting syntax diagram*

Listing 7-16 shows an example of using the `ON EXPLAIN` option. SQL*Plus executes the query, shows the query results, and displays the execution plan.

Listing 7-16. *Showing Query Results and Execution Plans*

```
SQL> set autotrace on explain
SQL> select ename from employees where empno < 7500;

ENAME
--------------
SMITH
ALLEN

Execution Plan
-----------------------------------------------------------------
 0      SELECT STATEMENT Optimizer=ALL_ROWS
        (Cost=2 Card=2 Bytes=20)
 1  0     TABLE ACCESS (BY INDEX ROWID) OF 'EMPLOYEES' (TABLE)
          (Cost=2 Card=2 Bytes=20)
 2  1       INDEX (RANGE SCAN) OF 'E_PK' (INDEX (UNIQUE))
            (Cost=1 Card=2)

SQL>
```

From Listing 7-16, you can see that the optimizer decided to use the unique index `E_PK` for a range scan, and it chose to access the `EMPLOYEES` table using the row identifiers resulting from the index range scan.

Listing 7-17 shows how you can use the `TRACEONLY STATISTICS` option to suppress the query results (you don't see the rows) and how you can produce a list of performance-related statement execution statistics. A detailed discussion of these statistics is not appropriate here, but you can see (for example) that no sorting was needed for this query, no data was read from disk (physical reads), and eight buffer cache block visits (consistent gets and db block gets) were needed.

Listing 7-17. *Showing Statistics Only*

```
SQL> set autotrace traceonly statistics
SQL> select * from employees;

14 rows selected.

Statistics
-------------------------------------------------------
          0  recursive calls
          0  db block gets
          8  consistent gets
          0  physical reads
          0  redo size
       1488  bytes sent via SQL*Net to client
        508  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
         14  rows processed

SQL> set autotrace off
SQL>
```

■**Note** If you use `AUTOTRACE TRACEONLY EXPLAIN`, the SQL statement is *not* executed. This is because you ask for only an execution plan, *not* for statement results and *not* for execution statistics.

7.7 Sequences

Information systems often use monotonically increasing sequence numbers for primary key columns, such as for orders, shipments, registrations, or invoices. You could implement this functionality with a small secondary table to maintain the last/current value for each primary key, but this approach is guaranteed to create performance problems in a multiuser environment. It is much better to use *sequences* in such cases.

Before we continue, there is one important thing you should know about sequences: sequence values can show gaps. That means that certain sequence values may disappear and never make it into the column they were meant for. The Oracle DBMS *cannot guarantee* sequences without gaps (we won't go into the technical details of why this is true). Normally, this should not be a problem. Primary key values are supposed to be unique, and increasing values are nice for sorting purposes, but there is no reason why you shouldn't allow gaps in the values. However, if the absence of gaps is a business requirement, you have no choice other than using a small secondary table to maintain these values.

Note If "absence of gaps" is one of your business requirements, then you probably have a poorly conceived business requirement. You should consider investing some time into reforming your business requirements.

Sequences can be created, changed, and dropped with the following three SQL commands:

```
SQL> create sequence <sequence name> ...
SQL> alter sequence <sequence name> ...
SQL> drop sequence <sequence name>;
```

Figure 7-15 shows the syntax diagram of the CREATE SEQUENCE command. The ALTER SEQUENCE command has a similar syntax.

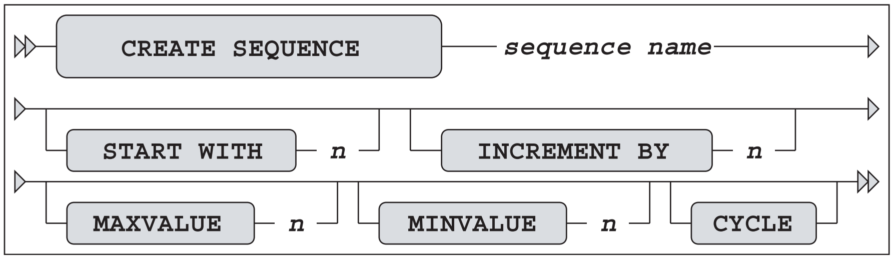

Figure 7-15. *CREATE SEQUENCE command syntax diagram*

A sequence definition may consist of a start value, increment value, minimum value, and maximum value. You can also specify whether the sequence generator should stop when reaching a boundary value, or CYCLE the sequence numbers within the minimum/maximum range. All sequence attributes are optional, as Figure 7-15 shows; they all have default values.

Each sequence has two pseudo columns: NEXTVAL and CURRVAL. The meaning of each of these columns is self-explanatory. Listing 7-18 shows how you can create and use a sequence DEPTNO_SEQ to generate department numbers, using the DUAL table. (Note that normally you would use sequence values in INSERT statements.)

Listing 7-18. *Creating and Using a Sequence*

```
SQL> create sequence deptno_seq
  2  start with 50 increment by 10;

Sequence created.

SQL> select deptno_seq.nextval, deptno_seq.currval from dual;

 NEXTVAL  CURRVAL
-------- --------
      50       50
```

```
SQL> select deptno_seq.currval from dual;

 CURRVAL
--------
      50

SQL> select deptno_seq.currval, deptno_seq.nextval from dual;

 CURRVAL  NEXTVAL
-------- --------
      60       60

SQL>
```

You can use CURRVAL multiple times, in different SQL statements, once you have selected NEXTVAL in an earlier statement, as shown in Listing 7-18. For example, in an order-entry system, you might select a sequence value with NEXTVAL to insert a new order, and then use the same value (CURRVAL) several times to insert multiple line items for that order.

Note the result of the last query in Listing 7-18. Since you select CURRVAL *before* NEXTVAL in the SELECT clause, you might expect to see the current value (50), followed by the next value (60), but apparently that is not the case. This behavior is based on the consistency principle that it doesn't matter in which order you specify the expressions in the SELECT clause of your queries, because you actually select those expressions *at the same time*. Try selecting NEXTVAL multiple times in the same SELECT clause and see what happens (the explanation is the same).

7.8 Synonyms

You can use the CREATE SYNONYM command to create synonyms for tables or views. Once created, you can use synonyms in all your SQL commands instead of "real" table (and view) names. For example, you could use synonyms for tables with very long table names.

Synonyms are especially useful if you are accessing tables from different schemas, not owned by yourself. Without synonyms, you must explicitly prefix those object names with the schema name and a period. The Oracle data dictionary is a perfect example of synonym usage. You can simply specify the data dictionary view names in your queries, without any prefix, although you obviously don't own those data dictionary objects.

Synonyms are a "convenience" feature. They don't provide any additional privileges, and they don't create security risks. They just save you some typing, and they also allow you to make your applications schema-independent.

Schema-independence is important. By using synonyms, your applications don't need to contain explicit schema names. This makes your applications more flexible and easier to maintain, because the mapping to physical schema and object names is in the synonym definitions, separated from the application code.

Figure 7-16 shows the syntax diagram for the `CREATE SYNONYM` command.

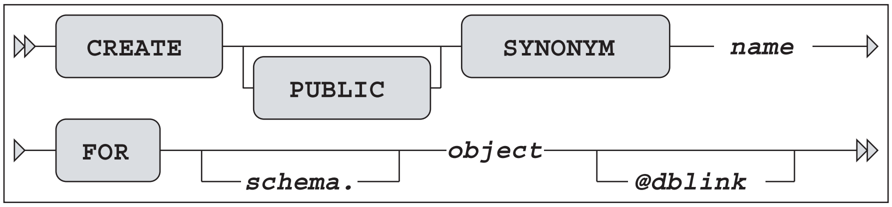

Figure 7-16. *CREATE SYNONYM command syntax diagram*

Oracle supports public and private synonyms, as you can see in Figure 7-16. By default, synonyms are private. You need to specify the `PUBLIC` keyword to create public synonyms. All database users can *use* public synonyms, but you need DBA privileges to be able to *create* them. The synonyms for the data dictionary objects are examples of public synonyms. Anyone can create private synonyms, but only their owners can use them.

■ **Caution** Although synonyms are useful, they can also cause performance problems. In particular, public synonyms are known to cause such problems. For further details, go to Steve Adams's web site (`http://www.ixora.com.au`) and search for "avoiding public synonyms."

Listing 7-19 shows how you can create a synonym, how the synonym shows up in the data dictionary views `CAT` and `USER_SYNONYMS`, and how you can drop a synonym.

Listing 7-19. *Creating and Dropping a Synonym*

```
SQL> create synonym e for employees;
Synonym created.

SQL> describe e
Name                     Null?    Type
------------------------ -------- ------------
EMPNO                    NOT NULL NUMBER(4)
ENAME                    NOT NULL VARCHAR2(8)
INIT                     NOT NULL VARCHAR2(5)
JOB                               VARCHAR2(8)
MGR                               NUMBER(4)
BDATE                    NOT NULL DATE
MSAL                     NOT NULL NUMBER(6,2)
COMM                              NUMBER(6,2)
DEPTNO                            NUMBER(2)

SQL> select * from cat;
```

```
TABLE_NAME           TABLE_TYPE
-------------------- -----------
EMPLOYEES            TABLE
DEPARTMENTS          TABLE
SALGRADES            TABLE
COURSES              TABLE
OFFERINGS            TABLE
REGISTRATIONS        TABLE
HISTORY              TABLE
DEPTNO_SEQ           SEQUENCE
E                    SYNONYM

SQL> select synonym_name, table_owner, table_name
  2  from   user_synonyms;

SYNONYM_NAME         TABLE_OWNER TABLE_NAME
-------------------- ----------- ----------------
E                    BOOK        EMPLOYEES

SQL> drop synonym e;
Synonym dropped.

SQL>
```

Synonyms are often used in distributed database environments to implement full data independence. The user (or database application) does not need to know where (in which database) tables or views are located. Normally, you need to specify explicit database links using the at sign (@) in the object name, but synonyms can hide those database link references.

7.9 The CURRENT_SCHEMA Setting

The ALTER SESSION command provides another convenient way to save you the effort of prefixing object names with their schema name, but without using synonyms. This is another "convenience" feature, just like synonyms.

Suppose the demo schema SCOTT (with the EMP and DEPT tables) is present in your database, and suppose you are currently connected as database user BOOK. In that situation, you can use the ALTER SESSION command as shown in Listing 7-20.

Listing 7-20. *The CURRENT_SCHEMA Setting*

```
SQL> alter session set current_schema=scott;
Session altered.

SQL> show user
USER is "BOOK"

SQL> select * from dept;
```

```
  DEPTNO DNAME          LOC
-------- -------------- -------------
      10 ACCOUNTING     NEW YORK
      20 RESEARCH       DALLAS
      30 SALES          CHICAGO
      40 OPERATIONS     BOSTON

SQL> alter session set current_schema=book;
Session altered.

SQL>
```

You can compare the CURRENT_SCHEMA setting in the database with the change directory (cd) command at the operating system level. In a similar way, it allows you to address all objects locally.

Again, this does not change anything with regard to security and privileges. If you really want to assume the identity of a schema owner, you must use the SQL*Plus CONNECT command, and provide the username/schema name and the corresponding password.

7.10 The DROP TABLE Command

You can drop your tables with the DROP TABLE command. Figure 7-17 shows the syntax diagram for the DROP TABLE command.

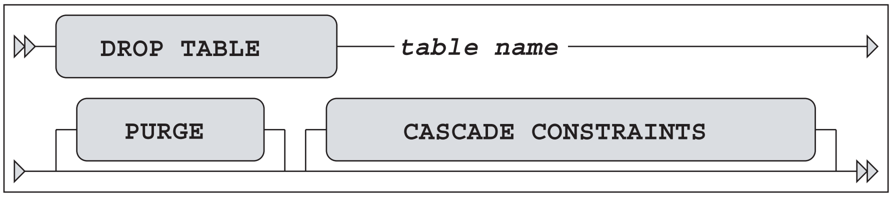

Figure 7-17. *DROP TABLE command syntax diagram*

Unless you have specific system privileges, you cannot drop tables owned by other database users. Also, you cannot roll back a DROP TABLE command. As you've learned in previous chapters, this is true for all DDL statements (CREATE, ALTER, and DROP).

"*Errare humanum est,*" as the Romans said. Because human errors occur occasionally, Oracle Database 10*g* introduced the concept of the database *recycle bin*. By default, all dropped tables (and their dependent objects) initially end up in the recycle bin. You can query the recycle bin using the [USER_]RECYCLEBIN view, as shown in Listing 7-21. To make sure we start with an empty recycle bin, we begin the experiment with a PURGE command.

Listing 7-21. *Dropping Tables and Querying the Recycle Bin*

```
SQL> purge recyclebin;
Recyclebin purged.

SQL> drop table history;
Table dropped.

SQL> select object_name, original_name, droptime
  2  from   recyclebin;

OBJECT_NAME                    ORIGINAL_NAME          DROPTIME
------------------------------ ---------------------- -------------------
BIN$mlRH1je9TBOeVEUhukIpCw==$0 H_PK                   2004-07-01:20:22:23
BIN$EETkZCYORSKCR3BhtF9cJw==$0 HISTORY                2004-07-01:20:22:23

SQL>
```

As you can see, the objects are renamed, but the original names are kept as well. There is one entry for the HISTORY table and one entry for the primary key index. You can recover tables (and optionally rename them) from the recycle bin by using the FLASHBACK TABLE command:

```
SQL> flashback table history to before drop
  2  [rename to <new name>];
Flashback complete.

SQL>
```

Caution There is no guarantee the FLASHBACK TABLE command always succeeds. The recycle bin can be purged explicitly (by a database administrator) or implicitly (by the Oracle DBMS).

If you want to drop a table and bypass the recycle bin, you can use the PURGE option of the DROP TABLE command, as shown in Figure 7-17.

If you drop a table, you implicitly drop certain dependent database objects, such as indexes, triggers, and table privileges granted to other database users. You also invalidate certain other database objects, such as views and packages. Keep this in mind during database reorganizations. To re-create a table, it is *not* enough to simply issue a CREATE TABLE command after a DROP TABLE command. You need to reestablish the full environment around the dropped table.

If you issue a DROP TABLE command, you may get the following error message if other tables contain foreign key constraints referencing the table that you are trying to drop:

```
ORA-02449: unique/primary keys in table referenced by foreign keys
```

Try to drop the EMPLOYEES table, and see what happens. You can solve this problem by using the CASCADE CONSTRAINTS option, as shown in Figure 7-17. Note, however, that this means that all offending foreign key constraints are dropped, too.

7.11 The TRUNCATE Command

The TRUNCATE command allows you to delete all rows from a table. Figure 7-18 shows the syntax diagram for the TRUNCATE command.

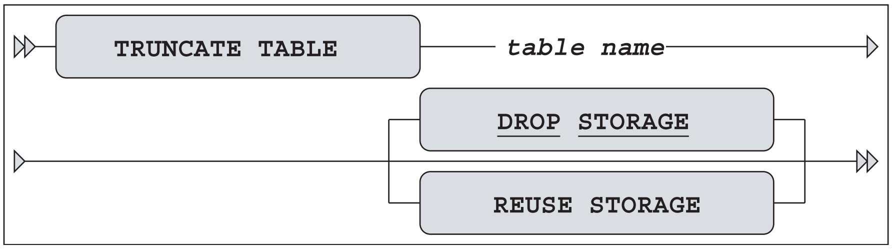

Figure 7-18. *TRUNCATE command syntax diagram*

The default behavior is DROP STORAGE, as indicated by the underlining in Figure 7-18.

Compared with DROP TABLE (followed by a CREATE TABLE), the big advantage of TRUNCATE is that all related indexes and privileges survive the TRUNCATE operation.

This command has two possible advantages over the DELETE command: the performance (response time) is typically better for large tables, and you can optionally reclaim the allocated space. However, there is a price to pay for these two advantages: you cannot perform a ROLLBACK to undo a TRUNCATE, because TRUNCATE is a DDL command. The Oracle DBMS treats DDL commands as single-statement transactions and commits them immediately.

7.12 The COMMENT Command

The COMMENT command allows you to add clarifying (semantic) explanations about tables and table columns to the data dictionary. Figure 7-19 shows the syntax diagram for this command.

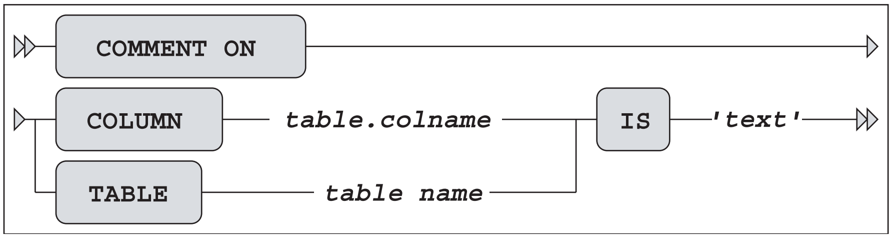

Figure 7-19. *COMMENT command syntax diagram*

Listing 7-22 shows how you can use the COMMENT command to add comments to the data dictionary for a table (SALGRADES) and a column (EMPLOYEES.COMM), and how you can retrieve that information from the data dictionary.

Listing 7-22. *Adding Comments to Columns and Tables*

```
SQL> comment on table salgrades
  2  is     'Salary grades and net bonuses';
Comment created.

SQL> comment on column employees.comm
  2  is     'For sales reps only';
Comment created.

SQL> select comments
  2  from   user_tab_comments
  3  where  table_name = 'SALGRADES';

COMMENTS
-------------------------------------------
Salary grades and net bonuses

SQL> select comments
  2  from   user_col_comments
  3  where  table_name  = 'EMPLOYEES'
  4  and    column_name = 'COMM';

COMMENTS
-------------------------------------------
For sales reps only

SQL>
```

7.13 Exercises

The following exercises will help you to better understand the concepts described in this chapter. The answers are presented in Appendix D.

1. Listing 7-5 defines the constraint E_SALES_CHK in a rather cryptic way. Formulate the same constraint without using DECODE and NVL2.
2. Why do you think the constraint E_DEPT_FK (in Listing 7-7) is created with a separate ALTER TABLE command?

3. Although this is not covered in this chapter, try to come up with an explanation of the following phenomenon: when using sequences, you cannot use the pseudo column CURRVAL in your session without first calling the pseudo column NEXTVAL:

```
SQL> select deptno_seq.currval from dual;
select deptno_seq.currval from dual
       *
ERROR at line 1:
ORA-08002: sequence DEPTNO_SEQ.CURRVAL is not yet defined in this session

SQL>
```

4. Why is it better to use sequences in a multiuser environment, as opposed to maintaining a secondary table with the last/current sequence values?

5. How is it possible that the EVALUATION column of the REGISTRATIONS table accepts null values, in spite of the constraint R_EVAL_CHK (see Listing 7-11)?

6. If you define a PRIMARY KEY or UNIQUE constraint, the Oracle DBMS normally creates a unique index under the covers (if none of the existing indexes can be used) to check the constraint. Investigate and explain what happens if you define such a constraint as DEFERRABLE.

7. You can use function-based indexes to implement "conditional uniqueness" constraints. Create a unique function-based index on the REGISTRATIONS table to check the following constraint: employees are allowed to attend the OAU course only once. They may attend other courses as many times as they like. Test your solution with the following command (it should fail):

```
SQL> insert into registrations values (7900,'OAU',trunc(sysdate),null);
```

Hint: You can use a CASE expression in the index expression.