CHAPTER 13

Managing Tables

This chapter shows you how to use SQL commands to create tables and indexes, define primary and foreign keys, and define other constraints. Some consider the command-line approach as old fashioned, preferring instead to use GUI-based tools. Don't worry, this chapter also shows you how to create tables using Enterprise Manager's Schema Manager application.

All the SQL commands shown in this chapter are also listed in Appendix A, "SQL Statement Reference." Because the commands are so complex, relevant examples are given here, but the full syntax is left to Appendix A.

Understanding Tables

Tables are the cornerstone of a relational database, and they typically consist of the following major elements:

- **Columns.** Define the data that may be stored in the table
- **Constraints.** Restrict the data that may be stored in a table, usually for the purpose of enforcing business rules
- **Indexes.** Allow fast access to a table's data based on values in one or more columns

  Oracle also uses indexes as a mechanism for enforcing unique and primary key constraints

Oracle is even more than a relational database. It is an object-relational database. In addition to the elements just described, Oracle tables may contain objects, arrays, and nested tables.

Chapter 29, "Using Oracle8i's Object Features," discusses these exciting features: objects, arrays, and nested tables.

Columns

Columns are mandatory elements of a table definition. You can define a table without indexes, and without constraints, but you can't define a table without columns. When creating columns, you'll want to consider names, datatypes, and null values.

Column Names

The rules for naming columns are the same as for any other object, including tables:

- ✦ Column names must begin with a letter.
- ✦ Column names may be up to 30 characters long.
- ✦ After the first letter, column names may consist of any combination of letters, digits, pound signs (#), dollar signs ($), and underscores (_).
- ✦ Oracle automatically converts all column names to uppercase unless they are enclosed within quotes.

Column names that are enclosed within quotes may contain other special characters besides those listed here, but creating such names usually ends up causing more trouble than it's worth.

Column names are not normally case-sensitive, unless they are enclosed in quotes. You'll find that it's usually best to avoid case-sensitive column names, although the ability to use them does come in handy when converting databases to Oracle from Microsoft Access, where mixed-case column names, and spaces in column names, are the rule.

Oracle8i Datatypes

Columns are defined during the process of creating a new table in the database. After naming the column, you must specify in Oracle what kind of data goes into the column. You do this by specifying a *datatype* for the column. Table 13-1 lists the datatypes available when you define columns in Oracle8i.

Table 13-1
Oracle8i Datatypes

Datatype	*Length*	*Description*
BFILE	0–4GB	A pointer to a binary file stored outside the database. The maximum supported file size is 4GB.

Datatype	*Length*	*Description*
BLOB	0–4GB	A binary large object. The maximum supported size is 4GB.
CHAR(*length*)	1–2000	A string with a fixed length. The default length is 1. Prior to the release of Oracle8, the maximum length of a CHAR field was 255.
CHARACTER(length)	1–2000	A string with fixed length. Provided for ANSI compatibility, it is the same as CHAR.
CHAR VARYING (*length*)	1–4000	A variable-length string. It is the same as CHARACTER VARYING.
CHARACTER VARYING (length)	1–4000	A variable-length string type provided for ANSI compatibility. It is the same as VARCHAR2.
CLOB	0–4GB	A character-based large object. It has a 4GB maximum size.
DATE	N/A	A date value. Valid dates range from January 1, 4712 B.C., to December 31, A.D. 4712. In Oracle, DATE fields always include the time in hours, minutes, and seconds.
DECIMAL (*precision*,*scale*)	precision: 1–38 scale: -84–127	A decimal datatype provided for ANSI compatibility. It is the same as NUMBER.
DOUBLE PRECISION	126 binary digits	A numeric datatype provided for ANSI compatibility. It is the same as FLOAT.
FLOAT(bdigits)	1–126	A floating-point type provided for ANSI compatibility. It is the same as NUMBER(bits). The default precision is 126.
INT	38 digits	An integer type provided for ANSI compatibility. It is the same as NUMBER(38).
INTEGER	38 digits	An integer type provided for ANSI compatibility. It is the same as NUMBER(38).
LONG	0–2GB	A large text string. The LONG datatype is obsolete. Use CLOB or BLOB instead for all new designs.

Continued

Table 13-1 *(continued)*

Datatype	*Length*	*Description*
LONG RAW	0–2GB	Raw binary data. Use BLOB instead.
MLSLABEL	N/A	A binary format of a label used on a secure operating system. This datatype is used only with Trusted Oracle.
NATIONAL CHAR (length)	1–2000	A national language type provided for ANSI compatibility. It is the same as NCHAR.
NATIONAL CHARACTER (length)	1–2000	A national language type provided for ANSI compatibility. It is the same as NCHAR.
NATIONAL CHAR VARYING (length)	1–4000	A national language type provided for ANSI compatibility, and which is the same as NVARCHAR2.
NATIONAL CHARACTER VARYING (length)	1–4000	A national language type provided for ANSI compatibility. It is the same as NVARCHAR2.
NCHAR(*length*)	1–2000	A fixed-length character string using the national character set. The NCHAR type has the same attributes as CHAR, except that it stores characters that depend on a national character set. Oracle8i supports many languages this way. The length is in bytes. In some character sets, one character may be represented by more than one byte.
NCLOB	0–4GB	A character-based large object. It uses a national character set.
NUMBER (*precision*,*scale*)	precision: 1–38 scale: -84–127	A number. The precision indicates the number of digits, up to 38. The scale indicates the location of the decimal point. For example, a column of type NUMBER (9,2) would store seven digits to the left of the decimal, and two to the right, for a total of nine digits.
NUMBER(*bdigits*)	1–126 binary digits	A floating-point number with up to 126 binary digits of precision. The default precision is 126.

Datatype	*Length*	*Description*
NUMBER	38 digits	A floating-point number. When no precision and scale are specified, a declaration of NUMBER results in a floating-point value with 38 digits of precision.
NVARCHAR2(*length*)	1–4000	A variable-length character string using the national character set. The NVARCHAR2 datatype has the same attributes as VARCHAR2, except that it stores characters using the national language character set. The length is in bytes, not characters.
RAW(*length*)	1–2000	Raw binary data. You should consider using BLOB instead.
REAL	63 binary digits	A real number. This is provided for ANSI compatibility and is the same as FLOAT(63).
REF	N/A	A pointer. It points to a particular instance of an Oracle8i object type.
ROWID	N/A	A rowid type. It allows you to store row IDs in a table.
SMALLINT	38 digits	Provided for ANSI compatibility. It is the same as NUMBER(38).
UROWID	N/A	Allows you to store universal row IDs, such as those that are used with index-organized tables (IOTs).
VARCHAR(*length*)	1–4000	Same as VARCHAR2. Oracle recommends using VARCHAR2 instead.
VARCHAR2(*length*)	1–4000	A text string of variable length. You must specify the maximum length when defining the column. Prior to the release of Oracle8, the maximum length of a VARCHAR2 was 2000 bytes.

The most common datatypes are VARCHAR2, DATE, and variations of NUMBER. Oracle rarely uses the datatypes provided for ANSI compatibility, and if they are used, Oracle converts them to the corresponding Oracle datatype.

Constraints

Constraints are rules about your data that you define at the database level. Constraints are declarative. You don't specify the process for enforcing a rule; you simply specify what that rule is to be. Oracle supports these types of constraints:

- ✦ Primary key
- ✦ Unique key
- ✦ Foreign key
- ✦ Check

Primary key constraints identify the set of columns whose values uniquely identify a record in a table. Oracle won't allow two records to have the same primary key value. For example, the `ID_NO` column is the primary key of the `AQUATIC_ANIMAL` table. Because `ID_NO` has been defined as the primary key, Oracle will not allow two records to have the same value in that column. Furthermore, primary key columns become required columns — they can never be null.

Unique key constraints identify sets of columns that must be unique for each row in a table. Unique key constraints are similar to primary key constraints, and they often represent alternative primary key choices. The one difference between a unique key and a primary key is that columns in a unique key may be null.

When you create a primary key or unique key constraint on a table, Oracle will create a unique index to enforce that constraint. The name of the index will match the name that you give the constraint.

Foreign key constraints are used to link two tables that contain related information. They are most often used in parent-child relationships, such as the one between the `TANK` table and the `AQUATIC_ANIMAL` table. A foreign key constraint can be placed on the `AQUATIC_ANIMAL` table, requiring that any `TANK_NO` value match a record in the `TANK` table. In this way, you can prevent erroneous tank numbers from being entered in the `AQUATIC_ANIMAL` table. Foreign keys must be linked to a corresponding primary or unique key.

Check constraints allow you to define an arbitrary condition that must be true before a row can be saved in a table. The most common check constraint is the `NOT NULL` constraint, which requires that a column have a value. You can supply almost any arbitrary expression for a check constraint, provided that it returns a value of `TRUE` or `FALSE`.

Indexes

Indexes on tables function much like the index in a book. They allow Oracle to quickly zero in on the data necessary to satisfy a query. Consider the TANK table, which contains the following information:

```
   TANK_NO TANK_NAME
---------- ----------------
         1 Dolphin Tank
         2 Penguin Pool
         3 Sea Lion Den
         4 Beer Tank
         ...
```

If you were to issue a SELECT statement looking for tanks named "Dolphin Tank," how would Oracle find the correct rows to return? One possible way would be to read each row in the table one by one, checking each to see if the name matched. With only four rows in the table, that's probably the fastest way, but what if there were a million rows to look through? With a million rows, it would take a long time to check each one to see if the name was "Dolphin Tank." An index on the TANK_NAME column would allow you to find the tank named "Dolphin Tank" quickly.

Chapter 14, "Managing Indexes," discusses the various index types that Oracle supports, showing you how to take advantage of them. With respect to creating and managing tables, be aware that Oracle creates indexes automatically to enforce primary and unique key constraints.

Creating Tables

When you create a relational table in an Oracle database, you need to do the following:

- ✦ Name the table.
- ✦ Name and define the columns within the table.
- ✦ Define constraints on the table.
- ✦ Specify physical properties for the table.

The last item, specifying physical properties, can be a large task in its own right, especially if you are creating a partitioned table, a clustered table, or a table containing large object types. Each of those topics is discussed in other chapters within this book.

There are two ways to create a table. One is to write a CREATE TABLE statement and execute it using a tool such as SQL*Plus. The other is to use Enterprise Manager's Schema Manager application.

Using the CREATE TABLE statement

Depending on what you are doing, the CREATE TABLE statement can be one of the most complex SQL statements to write. At a minimum, you should define the columns and specify a tablespace in which to store the table. For example:

```
CREATE TABLE aquatic_animal (
    ID_NO            NUMBER(10),
    TANK_NO          NUMBER(10),
    ANIMAL_NAME      VARCHAR2(30),
    BIRTH_DATE       DATE,
    DEATH_DATE       DATE,
    MARKINGS_DESCRIPTION VARCHAR2(30)
    ) TABLESPACE users;
```

This statement creates the AQUATIC_ANIMAL table used in the sample database and places it in the tablespace named USERS. The tablespace clause is optional. Omit it, and the table will be created as your default tablespace.

In a production setting, it's rare that you can store all your tables in the default tablespace. Production databases tend to have many tablespaces, and you'll find yourself including the TABLESPACE clause in most CREATE TABLE commands that you write.

Defining columns

Columns define the data that a table can store. As you've seen by looking at the previous CREATE TABLE statement, you define a column by providing a column name and a datatype. The exact syntax looks like this:

```
column_name datatype [DEFAULT default_value]
                     [[NOT] NULL]
                     column_constraint
```

The column name may be up to 30 characters long, and it must conform to the same rules used for naming tables and other objects. The datatype, unless you've defined your own, must be one of those listed in Table 13-1. The examples in this chapter focus on using the traditional scaler datatypes such as NUMBER, DATE, and VARCHAR2. Other chapters in this book delve into the more exotic areas of nested tables, large object types, and varying arrays.

When you define a column, you may define a default value for that column. You may also define one or more column constraints.

Default Values

Default values represent values that are stored in a column when no other value is supplied. They have meaning only when new rows are inserted, and you specify them using the DEFAULT clause in the column definition. The following example defines a default value for the MARKINGS_DESCRIPTION column:

```
MARKINGS_DESCRIPTION VARCHAR2(30) DEFAULT 'No unusual markings'
```

With this default value in place, if you insert a row into the AQUATIC_ANIMAL table without supplying a value for the MARKINGS_DESCRIPTION field, the default value would be stored automatically. For example:

```
SQL> INSERT INTO aquatic_animal
  2     (id_no, animal_name)
  3     VALUES (995,'Herman');

1 row created.

SQL> select id_no, markings_description
  2     FROM aquatic_animal
  3     WHERE id_no = 995;

  ID_NO MARKINGS_DESCRIPTION
--------- ------------------------------
  995 No unusual markings
```

Because no value is supplied for the MARKINGS_DESCRIPTION column, the default value is picked up and used instead. You can use default values for columns defined as NOT NULL to prevent errors caused when application programs insert a row without supplying a value for those columns.

Nullability

By default, Oracle doesn't require you to supply a value for every column when you are inserting a row into a table. Instead, columns that you omit are simply stored as null values. If you have important columns for which you always want the user to supply a value, you can use the NOT NULL keywords to indicate that. The ANIMAL_NAME column in the AQUATIC_ANIMAL table is a good candidate for a required field. Here's how you would make required fields out of both the ANIMAL_NAME column and the MARKINGS_DESCRIPTION column:

```
animal_name          VARCHAR2(30)
   CONSTRAINT animal_name_required NOT NULL,
markings_description VARCHAR2(30)
   DEFAULT 'No unusual markings'
   CONSTRAINT markings_description_required NOT NULL
```

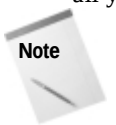
When you define a column as a NOT NULL column, you are actually defining a constraint on that column. A NOT NULL constraint is a specific implementation of a CHECK constraint. In this example, a NOT NULL constraint was defined for each field, and the CONSTRAINT keyword was used to give each constraint a name.

Column Constraints

You can define constraints at the column level by adding a CONSTRAINT clause to the column definition. The NOT NULL keywords just discussed actually represent one form of a constraint clause. You may also define primary keys, foreign keys, unique keys, and check constraints at the column level. The following example shows how you can define a primary key constraint for the AQUATIC_ANIMAL table:

```
ID_NO            NUMBER(10)
                 CONSTRAINT aquatic_animal_pk
                 PRIMARY KEY,
```

Defining constraints at the column level like this limits you somewhat, because you can't define a constraint over multiple columns. To do that, you need to use a table constraint. Since Oracle treats all constraints the same anyway, consider defining all your constraints at the table level.



The NOT NULL constraint is the constraint that makes sense at the column level. The ANSI standard supports the NOT NULL keywords, and people are quite used to using them.

Defining constraints

You can define constraints at the table level by including one or more constraint clauses along with your column definitions. Defining constraints on your tables is beneficial for the following reasons:

- ✦ Constraints prevent bad data from being saved.
- ✦ Constraints serve as a form of documentation. Foreign key and primary key constraints, for example, serve to document the relationships between tables. Reporting tools, such as Oracle Reports, can make use of that information.

You have four types of constraints to choose from: primary key, unique key, foreign key, and check.

Primary Key Constraints

You can use primary key constraints to define and document the columns in a table that can be depended upon to uniquely identify a record. One of the hallmarks of good relational design is that every table has a primary key. Primary keys enforce the following rules:

- ✦ Each column making up the key becomes a required column. Null values in primary key columns are not allowed. To do otherwise would mean that a primary key could not be depended upon to identify a record.
- ✦ No two rows in a table may have the same set of values for the primary key columns.

When you define a primary key constraint on a table, Oracle silently creates a unique index to support that constraint. Consequently, on all but the most trivial tables, when you create a primary key constraint, you need to give some thought to where that index will be stored and what it will be named. The following example demonstrates how to define a primary key constraint on the `AQUATIC_ANIMAL` table, including the clauses necessary to name the constraint, and how to specify the tablespace in which the underlying index is stored:

```
CREATE TABLE aquatic_animal (
    ID_NO            NUMBER(10),
    TANK_NO          NUMBER(10),
    ANIMAL_NAME      VARCHAR2(30),
    BIRTH_DATE       DATE,
    DEATH_DATE       DATE,
    MARKINGS_DESCRIPTION VARCHAR2(30),
    CONSTRAINT aquatic_animal_pk
         PRIMARY KEY (id_no)
         USING INDEX TABLESPACE indx
    ) TABLESPACE users;
```

Notice that the constraint clause follows immediately after the column definitions. You can mix table constraints in with the column definitions, but common practice is to list the constraints at the end.

When Oracle creates an index in support of a constraint, the index is named to match the constraint name. In this case, both the constraint and the index will be named `AQUATIC_ANIMAL_PK`. Naming your constraints is optional, but it's a good idea to give each constraint a unique name. Otherwise, Oracle generates a name. The names sometimes show up in error messages, and it's much easier to understand at a glance what a constraint named `AQUATIC_ANIMAL_PK` is all about than it is to guess at what `SYS_C001765` might be.

Having explicit names for constraints also makes it easier to write database maintenance and migration scripts. If you have four copies of a database, the system-generated constraint names will be different in each of the four. If you then need to perform maintenance related to constraints, it becomes impossible to write a generic script that works across all four databases.

The `USING INDEX` clause of a constraint definition allows you to specify a tablespace name and other parameters for the index that Oracle creates to enforce the constraint. The `USING INDEX` clause applies only to primary key and unique key constraints.

Unique Key Constraints

Unique key constraints are similar to primary key constraints. In fact, in many cases, unique keys represent alternative primary key choices. There is one major difference between unique and primary keys: Columns that form part of a unique key constraint may be null.

Listing 13-1 uses the AQUATIC_ANIMAL table and defines a unique key constraint on the animal name and tank number fields.

Listing 13-1: Defining a unique key constraint

```
CREATE TABLE aquatic_animal (
   ID_NO NUMBER(10),
   TANK_NO          NUMBER(10),
   ANIMAL_NAME      VARCHAR2(30),
   BIRTH_DATE       DATE,
   DEATH_DATE       DATE,
   MARKINGS_DESCRIPTION VARCHAR2(30),
   CONSTRAINT aquatic_animal_pk
         PRIMARY KEY (id_no)
         USING INDEX TABLESPACE indx,
   CONSTRAINT unique_name_tank
         UNIQUE (animal_name, tank_no)
         USING INDEX TABLESPACE indx
         STORAGE (INITIAL 50K NEXT 10K)
   ) TABLESPACE USERS;
```

As you can see, the definition of a unique key constraint is almost identical to that of a primary key. In this example, a storage clause has been added following the tablespace name to override the default storage parameters for the tablespace. As with a primary key constraint, Oracle will create an index to enforce the unique key constraint. The index name will match the constraint name.

When you create a unique constraint, you need to understand how Oracle treats null values when enforcing that constraint. As long as at least one column is not null, Oracle will enforce the constraint using the remaining values. If all columns are null, Oracle won't enforce the constraint. Consider the sequence of inserts into the AQUATIC_ANIMAL table that are shown in Table 13-2.

Table 13-2
The Effect of Nulls on a Unique Key Constraint

Sequence	*ID_NO*	*TANK_NO*	*ANIMAL_NAME*	*Results*
1	1	1	Flipper	Success.
2	2	1	Skippy	Success. The names are different.
3	3	1	Skippy	Failure. An existing record with the same tank number and animal name exists.
4	3	1		Success. No other record has a tank number of 1 with a null animal name.
5	4		Flipper	Success. No other record has a null tank number with an animal name of Flipper.
6	5		Flipper	Failure. An existing record already has a null tank number and an animal name of Flipper.
7	6			Success. Both tank number and animal name are null.
8	7			Success. Both tank number and animal name are null.

The key areas to focus on in Table 13-2 are sequences 5, 6, 7, and 8. Sequence numbers 5 and 6 show that when some columns are null, Oracle still enforces the constraint by looking for identical cases where the same columns are null. Sequences 7 and 8 show that any number of rows may be inserted when all columns listed in a unique constraint are null.

Foreign Key Constraints

Use foreign key constraints when you want to make certain that the values in a set of columns in one table exist in at least one row of another table. The `AQUATIC_ANIMAL` table provides a good example of the need for this. It contains a tank number for each animal in the table. The Seapark database also contains a `TANK` table, presumably with a list of valid tanks. You wouldn't want someone to enter an animal and place that animal in a tank that doesn't exist. You can prevent

that from ever happening by defining a foreign key on the AQUATIC_ANIMAL table, like this:

```
CREATE TABLE aquatic_animal (
...
CONSTRAINT assigned_tank_fk
        FOREIGN KEY (tank_no)
        REFERENCES tank
...
```

This constraint tells Oracle that every tank number value in the AQUATIC_ANIMAL table must also exist in the TANK table. How does Oracle know if a tank number is listed in the TANK table? It looks at the primary key. Oracle knows that the primary key of the TANK table is TANK_NO. Listing 13-2 illustrates how the foreign key is enforced.

Listing 13-2: **Enforcing a foreign key**

```
SQL> SELECT tank_no FROM tank;

TANK_NO
---------
        1
        2
        3
        4

SQL> INSERT INTO aquatic_animal
  2    (id_no, tank_no, animal_name)
  3    VALUES (970, 1, 'Sippy');

1 row created.

SQL> INSERT INTO aquatic_animal
  2    (id_no, tank_no, animal_name)
  3    VALUES (971, 5, 'Sappy');
insert into aquatic_animal
            *
ERROR at line 1:
ORA-02291: integrity constraint (SEAPARK.ASSIGNED_TANK_FK) violated -
parent key not found
```

The tank table contains entries for tanks numbered from 1 through 4. The first insert succeeds because it assigns the animal named Sippy to tank #1, a valid tank. The second insert fails because it assigns the animal named Sappy to tank #5, which is not a valid tank. In this way, a foreign key constraint prevents erroneous data from being entered into the database.

Foreign keys don't always have to relate to primary keys. They may also relate to unique keys. In that case, you must explicitly specify the matching columns in the parent table when defining the constraint. For example:

```
CREATE TABLE aquatic_animal (
...
CONSTRAINT assigned_tank_fk
        FOREIGN KEY (tank_no)
        REFERENCES tank (tank_no)
...
```

This example explicitly states that the TANK_NO column in the AQUATIC_ANIMAL table must match a value in the TANK_NO column of the TANK table. The column names don't need to match, although it is less confusing if they do match. They do need to be of the same datatype, and the referenced column(s) must represent either a primary key or a unique key.

Oracle doesn't need to create an index to enforce a foreign key constraint because primary and unique keys are already indexed. For performance reasons, though, consider indexing the foreign key fields in the child table. The reason for this is that if you ever delete a parent record (a tank record), Oracle must check to see if there are any child records (aquatic animal records) that reference the record being deleted. If there is no index on the foreign key fields in the child table, Oracle is forced to lock the table and read each record to see if any matching records exist.

If you know that you are never going to delete a parent record, fine. But if you do expect to delete parents, then index your foreign key. With respect to this example, you can index the TANK_NO column in the AQUATIC_ANIMAL table.

One last word on foreign key constraints: If at least one column listed in a foreign key constraint is null, Oracle will not check to see if a parent record exists. You can add the NOT NULL constraint to your foreign key fields if you always want to require a value.

Check Constraints

Check constraints allow you to define an arbitrary condition that must be true for each record in a table. The NOT NULL constraint is a form of check constraint where the condition is that a column must not be null. Earlier, you saw how to use the NOT NULL keywords to make MARKINGS_DESCRIPTION a required column. You can do the same thing by defining the following constraint:

```
CONSTRAINT markings_description_ck
    CHECK (markings_description IS NOT NULL)
```

It's probably better to define NOT NULL constraints in the normal fashion, using the NOT NULL keywords in the column definition.

Check constraints do have a place, though, as they allow you to define other useful conditions. Take the relationship between birth and death dates, for example. Most animals are born *before* they die. You can require that behavior by defining the following constraint:

```
CREATE TABLE aquatic_animal (
...
CONSTRAINT death_after_birth
        CHECK (death_date > birth_date),
...
```

Whenever a row is inserted into the AQUATIC_ANIMAL table, or whenever a row is updated, Oracle will evaluate the condition specified by the constraint. If the condition evaluates to TRUE (or to UNKNOWN), Oracle will allow the insert or update to occur.

When one of the columns in a check expression is null, that will cause the expression to evaluate to UNKNOWN, and Oracle will consider the constraint to be satisfied. The exception to this is when you explicitly write the expression to check for null values using IS NULL, IS NOT NULL, NVL, or some other construct designed to evaluate nulls.

Using Schema Manager to create a table

Rather than type a long command to create a table, you can use Enterprise Manager's Schema Manager to create a table using a GUI interface. To create a table using Schema Manager, follow these steps:

1. Run Schema Manager and connect to the database.
2. Right-click the Tables folder and select Create from the pop-up menu. See Figure 13-1.

   A Create Table window opens, and its tabbed interface allows you to enter the information needed to create the table.
3. Define the table's columns on the General tab.

   Figure 13-2 shows this tab filled out for the AQUATIC_ANIMALS table.
4. Switch to the Constraints tab and define the table's constraints.
5. Enter storage options and other options using the Storage and Options Tabs.
6. Click the Create button when you are ready to actually create the table.

The Schema Manager interface is fairly intuitive and easy to use. It functions just like any of the other Enterprise Manager applications. If you like, you can click the Show SQL button and watch Schema Manager build your CREATE TABLE statement for you as you fill in the tabs.

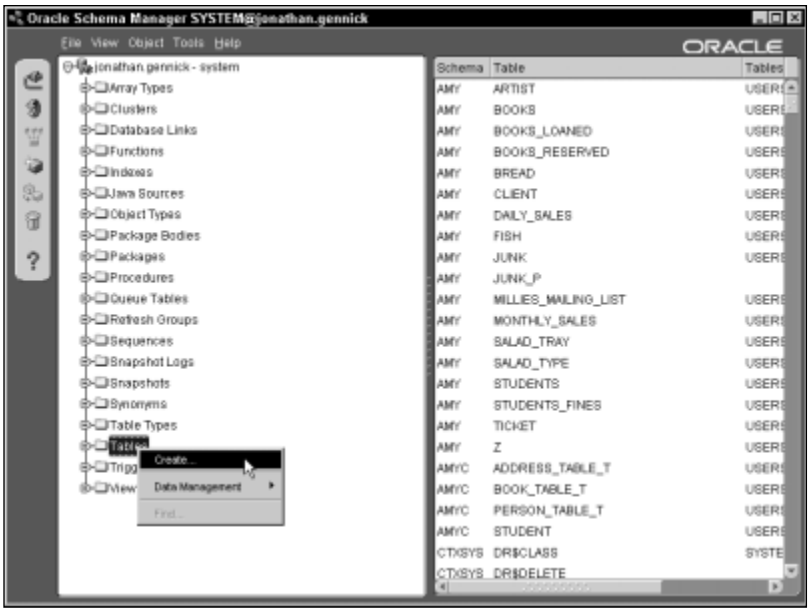


Figure 13-1: Creating a table in the Schema Manager window

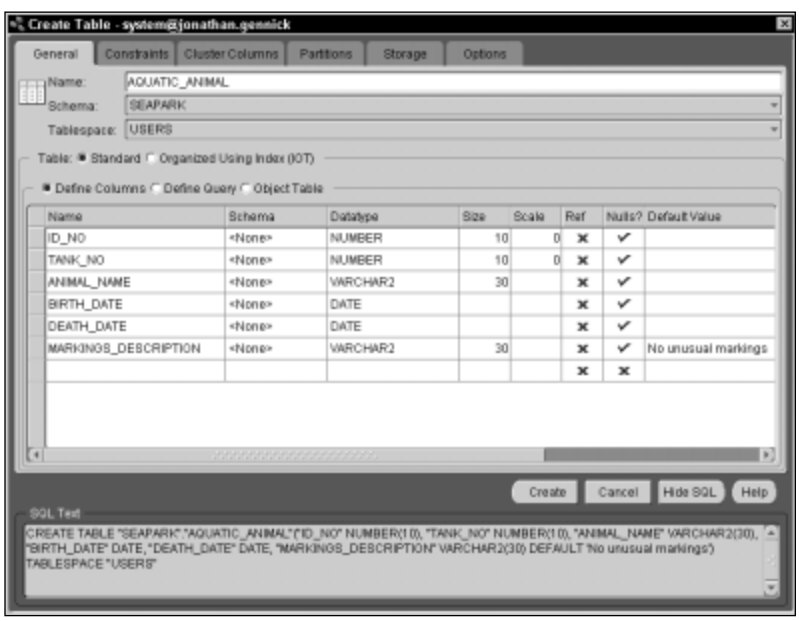


Figure 13-2: The General tab in the Create Table window

Altering Tables

From time to time, it's necessary to modify a table that you have created. In most cases, you will be able to use the ALTER TABLE command to make the changes that you need; however, you may find that some changes require you to drop and re-create the table.

Adding columns and constraints

You can easily add columns or constraints to a table by using the ALTER TABLE command as follows:

```
ALTER TABLE table_name
   ADD (column_or_constraint [,column_or_constraint...]);
```

The syntax you can use for columns or constraints that you wish to add to a table is identical to the syntax you use for creating a table. The following example adds a SEX column to the AQUATIC_ANIMAL table and constrains it to values of 'M' and 'F':

```
ALTER TABLE aquatic_animal
    ADD (animal_sex         CHAR,
         CONSTRAINT animal_sex_mf
            CHECK (animal_sex IN ('M','F'))
        );
```

When you add a column to the table, the value of that column is set to null for all rows that are currently in the table. For this reason, you can never add a new column with a NOT NULL constraint to a table that contains data. The very act of adding the column causes the constraint to be violated.

Another issue to be aware of is that if you add a column and specify a default value for the column at the time you add the column, Oracle will populate every row in the table with that default value. However, if you alter the table to add the column, and then alter the table and modify the column to add a default value, Oracle will only use the default value for rows added after that point.

Dropping columns and constraints

You can use the ALTER TABLE DROP command to drop columns and constraints. The syntax is slightly different for columns than it is for constraints.

The option of dropping a column is new with Oracle8i.

Dropping a Constraint

To drop a constraint, issue a command like this:

```
ALTER TABLE aquatic_animal
   DROP CONSTRAINT animal_sex_mf;
```

If you are dropping a primary key constraint or a unique key constraint and you have foreign key constraints that refer to the constraint that you are dropping, Oracle will disallow the drop unless you specify the `CASCADE` option. The following example demonstrates this option:

```
ALTER TABLE tank
   DROP CONSTRAINT tank_pk CASCADE;
```

Using the `CASCADE` option causes Oracle to drop not only the constraint that you named, in this case `TANK_PK`, but also any foreign key constraints that refer to it.

Dropping a Column

With the release of Oracle8i, Oracle implemented a feature long awaited by Oracle DBAs everywhere — the ability to drop a column from a table. Previously, the only way to do that was to drop the entire table and re-create it from scratch — a process made tedious by the need to save and restore the data, as well as by the need to disable and reenable constraints that referenced the table. Now, dropping a column is as simple as issuing a command like this:

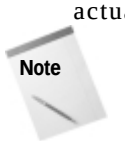
```
ALTER TABLE aquatic_animal
   DROP COLUMN animal_sex;
```

Use care when dropping a column on a large table. Oracle actually reads each row in the table and removes the column from each one. On a large table, that can be a time-consuming process.

If you want to drop a column in a hurry and you don't want to take the time to remove it from all the rows in the table, you can mark the column unused and delete it later. The command to mark a column unused is as follows:

```
ALTER TABLE aquatic_animal
   SET UNUSED (animal_sex);
```

Once you've marked a column as unused, it appears for all practical purposes as if the column is really gone. The only difference between setting a column unused and actually dropping the column is that the unused column still takes up space.



It's not easy to see the space freed by dropping a column. That's because the number of database blocks that the table uses doesn't change. Space is freed within the individual blocks, but the table as a whole is not compacted.

When you want to actually drop columns that you have marked as unused, you can issue a command like this:

```
ALTER TABLE aquatic_animal
   DROP UNUSED COLUMNS;
```

This command causes Oracle to go through and actually drop any columns that have been set unused.

Modifying columns

You can modify existing columns by issuing an `ALTER TABLE MODIFY` command and writing new column definitions for the columns that you want to change. For example, the following command changes the length of the `ANIMAL_NAME` column in the `AQUATIC_ANIMAL` table from its original maximum of 30 characters to a new maximum of 60 characters:

```
ALTER TABLE aquatic_animal
   MODIFY (animal_name    VARCHAR2(60));
```

You can use the same technique to add default values and constraints, such as a `NOT NULL` constraint, to the column.

Working with existing data

When you make changes to a table that contains existing data, you can run into a number of problems, including the following:

- ✦ The inability to add `NOT NULL` columns
- ✦ Incompatible datatypes when changing a column definition
- ✦ Preexisting records that violate foreign key constraints that you are trying to add
- ✦ Duplicate records that violate primary constraints that you are trying to add

Creating NOT NULL Columns

If you need to add a `NOT NULL` column to a table that already contains data, you can do so by breaking the task down into three steps:

1. Add the column without a `NOT NULL` constraint. Oracle will do this, and all existing rows will have the value for that column set to null.
2. Issue an `UPDATE` statement on the table that sets the column to a non-null value.
3. Modify the table and apply the `NOT NULL` constraint to the column.

Step 2 in this process requires some thought. You need some way to determine what the appropriate non-null value is for each row in the table. This may be a trivial task, and it may not be. You may find that step 2 becomes a significant process in its own right.

Working with Incompatible Datatypes

Sometimes you want to modify the datatype of a column, and the target datatype isn't compatible with the original. For example, Oracle won't let you convert a `NUMBER` column to a `VARCHAR2` column when that column contains data. Oracle will, however, let you freely change datatypes on empty columns (that is, when all values are null). This leads to the following possible approach of changing a column's datatype.

1. Create a temporary table with the column defined as you want it to be, using the new datatype. The temporary table should include all the primary key columns as well, from the table that you are changing. For example, if you need to change the `AQUATIC_ANIMAL` table's `TANK_NO` field to a character string, you can create this working table:

   ```
   CREATE TABLE aquatic_animal_work (
      ID_NO     NUMBER(10),
      TANK_NO   VARCHAR2(10));
   ```

2. Execute an `INSERT` statement that copies data from the table being changed to the work table. Code this `INSERT` statement to explicitly convert the column that you are changing. Consider this example:

   ```
   INSERT INTO aquatic_animal_work
      SELECT id_no, TRIM(TO_CHAR(tank_no))
      FROM aquatic_animal;
   ```

3. Set the column that you are changing to null. Do this for all rows in the table. Consider this example:

   ```
   UPDATE aquatic_animal
   SET tank_no = NULL;
   ```

4. Modify the column, changing the datatype to what you want it to be. Oracle should allow this because the column contains no data. Consider this example:

   ```
   ALTER TABLE aquatic_animal
   MODIFY (tank_no   VARCHAR2(10));
   ```

5. Move the saved column data from the work table back into the original table. Use the primary key to make sure that you get the correct value back in each row. Consider this example:

   ```
   UPDATE aquatic_animal aa
   SET tank_no = (SELECT tank_no
                  FROM aquatic_animal_work aaw
                  WHERE aaw.tank_no = aa.tank_no);
   ```

6. Drop the work table.

One critical point about using this approach is that you must do this when users are not accessing the table that you are changing.

Resolving Duplicate Primary or Unique Keys

If you are attempting to add a primary or unique key constraint to a table, and the existing data contains duplicate occurrences of the key values, you will receive an error like this one:

```
ORA-02437: cannot validate (SEAPARK.TANK_PK) - primary key violated
```

To resolve this issue, your first task is to get a list of offending rows. You can do that with a query like the following, which lists cases in the `TANK` table where two rows have the same tank number:

```
SELECT t1.tank_no, t1.tank_name
FROM tank t1
WHERE EXISTS (
   SELECT t2.tank_no
   FROM tank t2
   WHERE t2.tank_no = t1.tank_no
   GROUP BY t2.tank_no
   HAVING COUNT(*) > 1)
ORDER BY t1.tank_no;
```

Your next problem is deciding what to do with the duplicate rows. If they are truly duplicates, or if you are dealing with test data, you can do a mass delete by issuing a statement like the following:

```
DELETE
FROM tank t1
WHERE EXISTS (
   SELECT t2.tank_no
   FROM tank t2
   WHERE t2.tank_no = t1.tank_no
   GROUP BY t2.tank_no
   HAVING COUNT(*) > 1)
AND ROWID NOT IN (
   SELECT MIN(ROWID)
   FROM tank t3
   WHERE t3.tank_no = t1.tank_no);
```

You can depend on the `ROWID` psuedo-column to be unique for each record in the table, even if all the other columns are identical. This makes it useful for picking the one row to save out of each set of duplicates. The previous query arbitrarily chooses to save the row with the minimum `ROWID` value.

Another approach to this problem is to create an exceptions table, and use the `EXCEPTIONS INTO` clause with the constraint definition to cause Oracle to populate the exceptions table with the `rowids` of records that violate the constraint. You can

then work from that table to resolve the exceptions. Consider this example that shows how to use this technique:

```
ALTER TABLE aquatic_animal
  ADD ( CONSTRAINT tank_no_uk
           UNIQUE (tank_no)
           EXCEPTIONS INTO exceptions);
```

The attempt to add this constraint to the `aquatic_animal` table will fail because there are many animals in each tank. If even one row fails to meet the condition specified by the constraint, Oracle will not add it. However, since the `EXCEPTIONS INTO` clause is used, Oracle will write the `rowid`s of the offending rows into the table named exceptions. You can query the exceptions table, and join it to the `aquatic_animal` table, to see the list of rows that caused this constraint to fail. Consider this example:

```
SELECT id_no, tank_no, animal_name
FROM aquatic_animal, exceptions
WHERE exceptions.row_id = aquatic_animal.rowid
AND exceptions.table_name = 'AQUATIC_ANIMAL'
AND exceptions.constraint = 'TANK_NO_UK';
```

The exceptions table must be in a specific format. You can create it using an Oracle-supplied script named utlexcpt1.sql (utlexcpt.sql for releases prior to 8i), which you will find in the $ORACLE_HOME/rdbms/admin directory.

Resolving Nonexistent Foreign Keys

If you are trying to add a foreign key constraint to a table and parent keys don't exist for some of the records, you will get an error message like this:

```
ORA-02298: cannot validate (SEAPARK.ASSIGNED_TANK_FK) - parent keys not found
```

To find the offending rows, you can issue a query like this one:

```
SELECT id_no, tank_no
FROM aquatic_animal
WHERE NOT EXISTS (
   SELECT *
   FROM tank
   WHERE tank.tank_no = aquatic_animal.tank_no);
```

This query returns a list of all `AQUATIC_ANIMAL` records with tank numbers that don't exist in the `TANK` table.

The sample data, as created, does not contain any orphan tank numbers in the `AQUATIC_ANIMAL` table. If you want to execute this query and see results, delete one of the tank records first.

Of course, once you find the offending records, you have to decide what to do with them. That's the tough part. If you didn't care about them, you could convert the previous SELECT statement into a DELETE statement and delete them. A more reasonable approach might be to go in and correct them.

Changing storage parameters

When it comes to changing how the data for a table is physically stored, about the only way that you can make any significant changes is to drop and re-create the table. That makes sense when you begin to think about it. If, for example, you want to move a table's data from the USERS tablespace to a new tablespace named SEAPARK_DATA, you need to read the data from one tablespace and reinsert it into the other. No matter how you cut it, changing the way data is stored involves reading it and writing it back again.

If you need to make a significant change, such as moving a table from one tablespace to another, you can do so by following these steps:

1. Export the table. For more information about the Export utility, see Chapter 8, "Using Oracle8i's Export Utility."
2. Use the DROP TABLE statement to drop the table, including any constraints that reference the table.
3. Re-create the table, and don't forget to re-create foreign key constraints on other tables that reference the one that you are reorganizing.
4. Use the Import utility, with the IGNORE=Y option, to reload the table's data from the export file you created in step 1. For more information about the Import utility, see Chapter 9, "Using Oracle8i's Import Utility." You may also want to specify INDEXES=N so that you can re-create the indexes yourself after the import. That will generally give you better performance.
5. Re-create any indexes on the table. These will have been dropped when the table was dropped.

With large tables, this process can take some time. During the time that you are reorganizing the table, it won't be available to database users. This underscores the need for careful planning when you create your tables so that you don't need to reorganize them often.

Deleting Tables

When you no longer need a table, you can delete it using the DROP TABLE command. The syntax is simple and looks like this:

```
DROP TABLE [schema.]tablename [CASCADE CONSTRAINTS];
```

The CASCADE CONSTRAINTS option is necessary only if other tables have foreign key constraints that refer to the table you are dropping. The following example shows the AQUATIC_ANIMAL table being dropped:

```
SQL> DROP TABLE aquatic_animal CASCADE CONSTRAINTS;

Table dropped.
```

Of course, once a table is dropped, the data that it contained is gone too.

Using the Data Dictionary

The easiest way to look at the definition of a table is to use Schema Manager. Schema Manager's GUI interface allows you to easily browse column definitions, constraint definitions, and other information for any table in the database.

The SQL*Plus DESCRIBE command is also useful for viewing information about a table. Unlike Schema Manager, DESCRIBE tells you only about columns, their datatypes, and whether they can be null. The DESCRIBE command doesn't provide information about constraints, default values, or storage parameters. The following example shows the output that you get from DESCRIBE:

```
SQL> DESCRIBE aquatic_animal
 Name                           Null?      Type
 ------------------------------ -------- -------------
 ID_NO                          NOT NULL   NUMBER(10)
 TANK_NO                                   NUMBER(10)
 ANIMAL_NAME                               VARCHAR2(30)
 BIRTH_DATE                                DATE
 DEATH_DATE                                DATE
 MARKINGS_DESCRIPTION                      VARCHAR2(30)
```

If you don't have Schema Manager and you need more information than the DESCRIBE command provides, you can query Oracle's data dictionary views. In this section, you'll see how to use the following four views:

- **DBA_TABLES.** Returns information about each table in the database
- **DBA_CONSTRAINTS.** Returns information about constraints defined on a table
- **DBA_TAB_COLUMNS.** Returns information about the columns in a table
- **DBA_CONS_COLUMNS.** Returns information about the columns in a constraint

Typically, only database administrators have access to the DBA views because these views list information about every object in the database. If you aren't the DBA, you can replace DBA in the view names with either ALL or USER. For example, ALL_TABLES returns information about all tables that you have access to, and USER_TABLES returns information about all tables that you own.

Listing tables for a user

You can get a list of tables owned by any user in the database by querying the DBA_TABLES view. The DBA_TABLES view returns one row for each table in the database. The following query, executed from SQL*Plus, returns a list of all tables owned by SEAPARK:

```
SELECT table_name
FROM dba_tables
WHERE owner = 'SEAPARK'
ORDER BY table_name;
```

In addition to table names, you can also use the DBA_TABLES view to get information about a table's storage parameters, including its tablespace assignment.

Listing constraints on a table

The DBA_CONSTRAINTS view is quite useful because it provides information on the constraints defined for a table. There are four types of constraints, and you can use three different queries to see them. Primary and unique keys are similar enough that the same query works for both.

Check Constraints

Check constraints are the easiest to list because you can get everything you need from one view. The following query displays all the check constraints defined on the AQUATIC_ANIMAL table:

```
SELECT constraint_name,
       search_condition
FROM dba_constraints
WHERE owner = 'SEAPARK'
AND table_name = 'AQUATIC_ANIMAL'
AND constraint_type = 'C';
```

This query lists both the name of the constraint and the expression that must be satisfied. The SEARCH_CONDITION column returns the check expression. A constraint type of 'C' indicates that the constraint is a check constraint.

Primary and Unique Key Constraints

Returning information about primary key and unique key constraints is a bit more difficult than for check constraints because you have to join two views to get the list columns involved in the constraints. Primary key constraints are identified by a type of 'P', while unique key constraints have a type of 'U'. The following query lists primary and unique key constraints for the AQUATIC_ANIMAL table:

```
SELECT c.constraint_name,
       c.constraint_type,
       cc.column_name
```

```
FROM dba_constraints c,
     dba_cons_columns cc
WHERE c.owner='SEAPARK'
AND c.table_name = 'AQUATIC_ANIMAL'
AND c.owner = cc.owner
AND c.constraint_name = cc.constraint_name
AND c.constraint_type IN ('P','U')
ORDER BY c.constraint_type,
         c.constraint_name,
         cc.position;
```

The `ORDER BY` clause is necessary. This query will return one row for each column in a constraint. Sorting on the constraint name causes all columns for a constraint to be listed together. Further sorting by the `POSITION` field is done to get the columns to list in the proper order. The `DECODE` at the beginning of the `ORDER BY` clause is optional and causes any primary key constraint to be listed first, before any unique key constraints.

Foreign Key Constraints

Foreign key constraints are the most difficult to query for. Not only must you list the columns involved in the constraint, but you must also join `DBA_CONS_COLUMNS` to itself to determine the parent table and the matching list of columns in the parent table. Foreign key constraints are identified by a type code of `'R'`. The query shown in Listing 13-3 lists all foreign key constraints defined on the `AQUATIC_ANIMAL` table.

Listing 13-3: Listing a table's foreign key constraints

```
SELECT c.constraint_name,
       cc.column_name,
       rcc.owner,
       rcc.table_name,
       rcc.column_name
FROM dba_constraints c,
       dba_cons_columns cc,
       dba_cons_columns rcc
 WHERE c.owner='SEAPARK'
AND c.table_name = 'AQUATIC_ANIMAL'
AND c.constraint_type = 'R'
AND c.owner = cc.owner
AND c.constraint_name = cc.constraint_name
AND c.r_owner = rcc.owner
AND c.r_constraint_name = rcc.constraint_name
AND cc.position = rcc.position
ORDER BY c.constraint_name, cc.position;
```

The results are sorted by constraint name and column position so that the columns will be listed in the proper order.

Listing the columns in a table

The DBA_CONS_COLUMNS view returns information about the columns in a table. It's the same view that SQL*Plus queries when you issue the DESCRIBE command. For the most part, you're better off using DESCRIBE than trying to query this view. The exception to that is if you need to see the default value for a column. The DESCRIBE command won't provide you with that information. The following query lists all columns in the AQUATIC_ANIMAL table that have default values defined, and it shows you what those default values are:

```
SELECT column_name, data_default
FROM dba_tab_columns
WHERE owner = 'SEAPARK'
AND table_name = 'AQUATIC_ANIMAL'
AND data_default IS NOT NULL;
```

The DATA_DEFAULT column is a LONG column, which allows for values up to 2GB in size. By default, SQL*Plus will display only the first 80 characters. If you are dealing with long strings, you can issue the SET LONG command to increase the number of displayed characters. For example, SET LONG 160 causes SQL*Plus to display the first 160 characters of each LONG value.

Summary

In this chapter, you learned:

- ✦ Tables are composed of columns and constraints. Columns define what a table can store, while constraints help prevent the introduction of invalid data.
- ✦ Oracle supports a number of datatypes. The most common scaler types are NUMBER, VARCHAR2, DATE, and CHAR.
- ✦ Use the CREATE TABLE statement to create a new table. Be sure to specify a tablespace and to define a primary key.
- ✦ Oracle uses indexes to enforce primary key and unique key constraints. Be sure to assign these indexes to a tablespace when creating these types of constraints.
- ✦ To modify a table, use the ALTER TABLE command. Changing physical storage parameters may require a full export and import of the table. If the table contains data, you may have to work around that data to make your changes.

- If you don't have Enterprise Manager installed, you can still get information about a table from Oracle's data dictionary views. The `DBA_CONSTRAINTS` and `DBA_CONS_COLUMNS` views provide information about table constraints. The `DBA_TABLES` and `DBA_TAB_COLUMNS` views provide information about table and column definitions.
- If you are not the DBA, you won't have access to the DBA views. Use `ALL_TABLES`, `ALL_TAB_COLUMNS`, `ALL_CONSTRAINTS`, and `ALL_CONS_COLUMNS` instead.

✦ ✦ ✦