CHAPTER 15

Working with Views, Synonyms, and Sequences

In This Chapter

Managing views

Managing synonyms

Managing sequences

Views, synonyms, and sequences are relatively minor objects in terms of the effort you must expend to create and manage them. *Views* are nothing more than stored SELECT statements. Once you create them, Oracle allows you to treat a view as if it were another table. *Synonyms* represent alterna-tive names for a database object. The primary use of synonyms is to provide schema independence. *Sequences* are transaction-independent counters that are stored in the database and managed by Oracle. Their primary application is to generate unique keys for tables.

Views have applications for security and for convenience. In terms of security, views can limit the data that a user is allowed to see. In terms of convenience, views may encap-sulate complex SQL needed for reporting, reducing the amount of complexity that goes into generating the reports. Synonyms allow users to access tables owned by an applica-tion schema without having to preface each table name with the schema name. This chapter discusses views, synonyms, and sequences, and explains why you use each type as well as how to create and manage them.

Managing Views

To create and manage views, you first need to know what a view is. It also helps to understand how and why views are used and why they are important. A view is a stored SELECT statement that presents data in a more convenient form than you might find in your tables. You can encapsulate complex SQL statements into a view, relieving users from having to write those statements themselves. Consider the following view on the AQUATIC_ANIMAL table:

```
CREATE OR REPLACE VIEW animals AS
   SELECT id_no,
          animal_name name,
          TO_CHAR(birth_date,'dd-Mon-yyyy') birth_date
FROM aquatic_animal
WHERE death_date IS NULL;
```

Consider this a convenience view. It limits the results to only those animals who are still alive. It formats the date field to use a four-digit year, and it simplifies the title of the ANIMAL_NAME column to just NAME. Here's what your output will look like when you select from this view:

```
SQL> select * FROM animals;

      ID_NO NAME                           BIRTH_DATE
--------- ------------------------------ -----------
      100 Flipper                        01-Feb-1968
      105 Skipper                        01-Jan-1978
      112 Bopper                         11-Mar-1990
```

What really happens when you select from a view like this? Let's say that you issue the following SELECT statement against the view:

```
SELECT *
FROM animals
WHERE name = 'Skipper';
```

From a conceptual standpoint, when you query a view, Oracle first executes the view's query and then executes your query against the results. It would be as if you had written a SQL statement like this:

```
SELECT *
FROM (
SELECT id_no,
          animal_name name,
          TO_CHAR(birth_date,'dd-Mon-yyyy') birth_date
FROM aquatic_animal
WHERE death_date IS NULL
)
WHERE name = 'Skipper';
```

Conceptually, this is what happens. In some cases, if a view is complex enough, Oracle will actually execute the view this way. In practice, though, completely executing the view's query each time the view is referenced doesn't lead to good performance. When it can, the optimizer will do one of the following for queries against a view:

- ✦ Merge your query into the view's query
- ✦ Merge the view's query into your query

Believe it or not, the optimizer does consider these as two separate approaches. The result is that the query against the `ANIMALS` view for the animal named Skipper would be translated into something like this:

```
SELECT id_no,
       animal_name name,
       TO_CHAR(birth_date,'dd-Mon-yyyy') birth_date
FROM aquatic_animal
WHERE animal_name = 'Skipper'
AND death_date IS NULL;
```

Merging the two queries is a much more efficient means of processing a query against a view than to execute the view's query first, and Oracle will do this whenever possible. Remember the conceptual model, though. No matter how Oracle optimizes a query against a view, the results must match the conceptual model. When writing queries against views, you'll find it helpful to always think in terms of that model.

Determining when to use views

Views have a number of uses, including the following:

- ✦ **Security.** You can use views to limit a user's access to the rows in a table, to limit a user's access to columns in a table, and to restrict a user's ability to insert data into a table.
- ✦ **Convenience.** You can use views to encapsulate complex SQL queries, making report generation easier.
- ✦ **Consistency.** You can encapsulate standard reports into views. Users querying these views will get consistent results. Views can also insulate you from changes to the underlying tables.

Security

Views are often used to prevent a user from seeing all the data within a table. For example, you might limit a user to a certain subset of rows, or you might block out certain columns. Chapter 12, "Using Fine-Grained Access Control," contains an

example of views being used for security. When you use a view for security, you must take the following approach:

- ✦ Deny users access to the underlying table. You don't want the users going around the view and querying the table directly.
- ✦ Create a view that limits what a user can see.
- ✦ Grant users access to the view.

A key feature of views is that users don't need access to the underlying table to select from the views. Views were purposely designed this way so that they can be used as a way to limit a user's access to data. The following view, for example, limits a user to seeing only animal ID numbers and names:

```
CREATE VIEW animal_name AS
SELECT id_no,
       animal_name
FROM aquatic_animal;
```

In terms of the sample database, one could hardly consider an animal's birth date to be sensitive information. However, if you were dealing with an HR application, you might consider an employee's pay rate to be sensitive, and you could create a view that didn't contain that column.

Another, possibly security-related purpose of views, is to use them to limit what a user can insert into a table. You can do this by taking advantage of the `WITH CHECK OPTION`. Take a look at the following view, which returns a list of live animals:

```
CREATE OR REPLACE VIEW live_animals AS
SELECT *
FROM aquatic_animal
WHERE death_date IS NULL
WITH CHECK OPTION;
```

The keywords `WITH CHECK OPTION` prevent someone from using this view to insert any records that contain a death date. In other words, users are prevented from entering an already dead animal into the system. Users would also be prevented from entering a death date for any existing animals. The following example shows the error that you would get if you tried to insert a record that violated the view's conditions:

```
SQL> INSERT INTO live_animals
  2  (id_no, animal_name, death_date)
  3  VALUES (576, 'Scampi',
  4          TO_DATE('3-Sep-1999','dd-Mon-yyyy'));
INSERT INTO live_animals
            *
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation
```

You might consider this to be as much a data integrity issue as a security issue. Presumably, however, you would have a few select people who were allowed access directly to the table so that they could record the death of an animal.

If you have different classes of users, you can create many different views, one for each class. As the number of views increases, so does the effort needed to manage them. If you find yourself dealing with an unmanageable number of views, you might take a look at Chapter 12 to see if you can make use of Oracle8i's new fine-grained access control features.

Convenience

You can also use views for convenience. If you have complex queries on which reports are based, you can encapsulate those queries into views. That makes the programmer's job easier, and if he or she is using the same query repeatedly, using a view instead of replicating the query over and over reduces the risk of introducing an error.

Another way you can use convenience views is to make life easy for end users who are querying the database using ad-hoc query tools. In spite of the nice GUI interfaces that these tools support, querying a database still requires knowledge of how to deal with many subtle issues. Users need to know about nulls and their impact on true or false expressions. They need to understand outer joins vs. inner joins. There may be subtleties in the data or exceptions to the rule that the average user won't know about or understand. Instead of leaving your users to deal with these issues, you can create views that pull together the data that they most often require. Here's a good example:

```
CREATE OR REPLACE VIEW animal_caretakers AS
SELECT id_no,
       t.tank_no,
       animal_name,
       full_name caretaker_name
FROM aquatic_animal a, tank t, caretaker c
WHERE a.tank_no = t.tank_no(+)
AND t.chief_caretaker_name = c.caretaker_name (+)
AND a.death_date IS NULL;
```

This view returns information about animals and their caretakers. Note that it omits dead animals, because presumably the average user doesn't need to be concerned about animals that don't exist. The view also includes an outer join, so animals without a tank assignment will still be included. Both of these issues represent complexities that an end user might not think about.

The ANIMAL_CARETAKERS view shown in this section is a relatively simple view. In production systems, the queries behind views can be several pages long. Placing those queries into a view makes a huge difference in usability because the user no longer has to type in, or even think about, such a long query.

Consistency

If a great deal of specialized knowledge is required to properly query your tables, different users will tend to get different results even when producing ostensibly identical reports. You may have two users producing a list of animals in the Seapark facilities, but only one might remember to exclude those animals who have died. By creating a standardized set of views that take these subtleties into account, and by requiring ad-hoc reports to be written against those views, you promote consistency in results across the organization. For example, an organization can use this technique with its financial data so that reports produced by different departments will be consistent with one another.

Creating a view

If you've read this far into the chapter, you've already seen several examples showing how to create a view. To start with, you need three items:

- A name for the view
- A SQL query
- The `CREATE VIEW` system privilege

Once you have these three items, you can use the `CREATE VIEW` statement to create a view. Consider this example:

```
CREATE VIEW animal_checkup AS
   SELECT a.id_no,
          a.animal_name,
          c.checkup_date,
          c.checkup_type,
          c.doctor_name
   FROM aquatic_animal a, checkup_history c
   WHERE a.id_no = c.id_no;
```

You can specify some useful options when you create a view. These include the following:

- `OR REPLACE` — Allows the new view to replace an existing view of the same name. This saves you the trouble of dropping the old view first. These keywords should immediately follow `CREATE`: for example, `CREATE OR REPLACE`.
- `FORCE` — Creates the view even if the underlying tables don't exist or if you don't have access to those tables. However, the view will remain in an invalid state until the tables are created and you have been granted access to them. If you use this option, `FORCE` should precede the `VIEW` keyword: for example, `CREATE FORCE VIEW`.

The export utility generates `CREATE FORCE VIEW` statements to ensure that views get created on import, regardless of whether all objects referenced by a view have been created yet.

- `WITH READ ONLY` — Prevents all but query access to the view. These keywords follow the SQL query.
- `WITH CHECK OPTION` — Restricts users to inserting and updating rows that fit within the view. Consider using this option for views that are used for `UPDATES` and `INSERTS`.

This option may not work properly if the view contains a subquery.

If you prefer to use a GUI interface, you can use Schema Manager to create a view. Just follow these steps:

1. Start Schema Manager and log on to the database.
2. Right-click the Views folder and select Create from the pop-up menu.

   The Create View dialog box, shown in Figure 15-1, will open.
3. Fill in the fields of the General tab with the information necessary to create the view.

   Figure 15-1 shows the `ANIMAL_CHECKUP` view being created.
4. Click the create button.

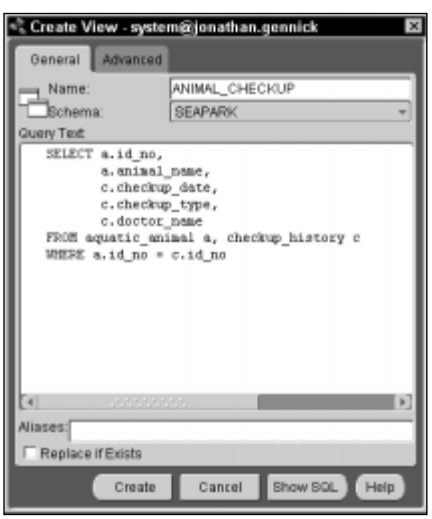


Figure 15-1: Using Schema Manager to create a view

Schema Manager doesn't really provide any advantages over the command-line option for creating views. You still need to type the SQL query. In fact, that little text box might complicate matters if your SQL query is particularly long.

You can use any valid SQL query in a view. The query can use Oracle's built-in SQL functions, column aliases, group functions, and so forth. To learn more about constructing queries, see Chapter 16, "Selecting Data with SQL."

Selecting from a view

Once you've created a view, you can select from it just as if it were a table. Consider this example:

```
SQL> SELECT * FROM animal_checkup;

   ID_NO ANIMAL_NAME  CHECKUP_D CHECKUP_TY  DOCTOR_NAME
------- ------------ --------- ---------- ------------------
   105 Skipper        03-SEP-99 ANNUAL      Justin Nue

1 row selected.
```

You can even join views and tables together. The following query joins the `ANIMAL_CHECKUP` view to the `CHECKUP` table and returns only those checkups for which a blood sample should have been obtained:

```
SELECT *
FROM animal_checkup, checkup
WHERE animal_checkup.checkup_type = checkup.checkup_type
AND blood_test_flag = 'Y';
```

While views and tables may be used interchangeably when selecting data, that isn't always the case when you are changing data.

Using views to manipulate data

Where possible, Oracle allows you to issue `UPDATE`, `INSERT`, and `DELETE` statements against a view. However, when it comes to views, these types of statements present a problem. All of these statements modify the data in the underlying table. To update or delete a row through a view, you need to be able to track the row in the view back to one row in a table underlying the view. This gets difficult when you are dealing with summarized data, columns based on expressions, unions, and certain types of joins.

Generally, if a view includes all columns from the underlying table and uses only a `WHERE` clause to restrict the rows returned, Oracle will allow you to update the view. Certain types of join views are also updateable, but the following rules apply:

- Only one of the tables represented in a view can be updated.

- ✦ The columns that you are updating must map back to a table whose primary key columns are all preserved in the view.
- ✦ If you are deleting from a join view, then the view must include primary key columns from only one of the tables involved in the join. The delete will apply to only that table.
- ✦ If you are inserting into a view, you must not reference nonupdatable columns. If the join view is created using the `WITH CHECK OPTION`, you won't be able to insert into it at all.

Join views that don't preserve the primary keys of at least one underlying table can never be updated. Other types of views are also never updated. These include the following:

- ✦ Views with set operators such as `INTERSECT`, `UNION`, and `MINUS`
- ✦ Views with `GROUP BY`, `CONNECT BY`, or `START WITH` clauses
- ✦ Views with group functions such as `AVG`, `SUM`, or `MAX`
- ✦ Views using the `DISTINCT` function

If you have any doubts about which views you can update or about which columns in a specific view you can update, you can query the `USER_UPDATABLE_COLUMNS` data dictionary view. For example, if you create a view named `TANK_ANIMALS_VIEW` that joins two tables, the following query will show which columns you can update:

```
SELECT column_name, updatable
FROM user_updatable_columns
WHERE table_name = 'TANK_ANIMALS_VIEW';
```

If you need to make a nonupdateable view updateable, you may be able to do that by writing some instead-of triggers. An *instead-of trigger* defines PL/SQL code to be invoked by Oracle in response to updates, inserts, and deletes issued against a view.

You can read about instead-of triggers in Chapter 24, "Using Procedures, Packages, Functions, and Triggers."

Altering a view

Oracle does implement an `ALTER VIEW` command, but the only operation that you can do with it is to recompile the view. To make any other changes, you must re-create the view.

Recompiling a View

Recompiling a view causes Oracle to check the underlying SQL statement to be sure that it is still valid. The syntax to use is as follows:

```
ALTER VIEW [schema.]view_name COMPILE;
```

Any time that you change an object referenced by a view's query, Oracle will mark that view as invalid. Before you can use the view again, it must be recompiled. You can either recompile it yourself, or leave Oracle to do it automatically. If you're in doubt about whether you have any views that need to be recompiled, you can get a list of invalid views by issuing this query:

```
SELECT object_name, status
FROM user_objects
WHERE object_type = 'VIEW'
AND status = 'INVALID'
```

You can even use SQL to automatically generate the needed `ALTER VIEW` commands to do the recompiles. Execute a query like the following using SQL*Plus, and spool the output to a file:

```
SELECT 'ALTER VIEW ' || object_name || ' COMPILE;'
FROM user_objects
WHERE object_type = 'VIEW'
AND status = 'INVALID'
```

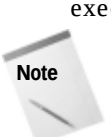
When you leave the recompiling for Oracle to do automatically, you run the risk that the view won't compile correctly. Explicitly compiling invalid views is a safer approach because it allows you to spot any errors up front, allowing you to fix them before the users encounter them.

Re-creating a View

Aside from recompiling a view, you must make any other changes by re-creating the view. Oracle makes this fairly easy: All you have to do is use `OR REPLACE` with the `CREATE VIEW` command. For example:

```
CREATE OR REPLACE VIEW animal_checkup AS
   SELECT a.id_no,
          a.animal_name,
          c.checkup_date,
          c.checkup_type,
          c.doctor_name
   FROM aquatic_animal a, checkup_history c
   WHERE a.id_no = c.id_no;
```

If you store your `CREATE VIEW` statements in text files, you can quickly make changes to a view by editing the view's `CREATE` statement in the file and then executing that file using SQL*Plus.



For convenience, the `CREATE OR REPLACE` syntax is valid even if the view does not already exist.

Using Schema Manager to Alter a View

You can use Schema Manager to modify a view in much the same way as you use it to modify a table. Simply follow these steps:

1. Start Schema Manager and log on to the database.
2. Open the Views folder and navigate to the view that you want to change.
3. Right-click the view name and select Edit from the pop-up menu.

   Schema Manager opens an Edit View window for you.
4. Make your changes to the view by editing the fields in the window.
5. Click the OK button.

If you click the Show SQL button while editing a view in Schema Manager, you will see that Schema Manager makes the changes by writing `CREATE OR REPLACE` statements to re-create the view.

Removing a view

You can use the `DROP VIEW` command to remove a view from the database. The following statement, for example, removes the view named `ANIMAL_CHECKUP`:

```
DROP VIEW animal_checkup;
```

You can use Schema Manager to drop a view by right-clicking the view and selecting Remove from the pop-up menu.

Using views with the data dictionary

Two data dictionary views are useful when it comes to managing views. The `DBA_OBJECTS` view allows you to query for views that are currently in an invalid state and that need to be recompiled. The `DBA_OBJECTS` view is similar to `USER_OBJECTS`, which was discussed previously in the section "Altering a View."

The `DBA_VIEWS` view is another useful view. You can use `DBA_VIEWS` to get a list of views owned by a user and to get the query behind a view. If you're not a DBA, you can use `ALL_VIEWS` instead of `DBA_VIEWS`.

Listing Views Owned by a User

You can query the `DBA_VIEWS` view to generate a list of all views owned by any particular user. The query in the following example returns a list of views owned by `SEAPARK`:

```
SQL> SELECT view_name
  2  FROM all_views
```

```
  3  WHERE owner='SEAPARK';

VIEW_NAME
------------------------------
ANIMALS
ANIMAL_CARETAKERS
ANIMAL_CHECKUP
ANIMAL_NAME
CHECKUP_TYPE_COUNT
LIVE_ANIMALS
USER_TANK
```

If you're interested only in your own views, select from `USER_VIEWS` instead and omit the `WHERE` clause. That will save you some typing.

Displaying the Query behind a View

The query behind a view is returned by the column named `TEXT` in the `DBA_VIEWS` view. Listing 15-1 shows the query being returned for the view named `ANIMAL_CHECKUP`.

Listing 15-1: Retrieving a view's query

```
SQL> SET LONG 2000
SQL> COLUMN text WORD_WRAPPED
SQL>
SQL> SELECT text
  2  FROM DBA_VIEWS
  3  WHERE owner = 'SEAPARK'
  4  AND view_name = 'ANIMAL_CHECKUP';

TEXT
------------------------------------------------
SELECT a.id_no,
          a.animal_name,
          c.checkup_date,
          c.checkup_type,
          c.doctor_name
   FROM aquatic_animal a, checkup_history c
   WHERE a.id_no = c.id_no


1 row selected.
```

The datatype of the `TEXT` column is `LONG`. By default, SQL*Plus displays only the first 80 characters of a `LONG` column. Unless you write very short queries for your

views, that won't do you any good. The SET LONG 2000 command tells SQL*Plus to display up to 2000 characters of the LONG value. For complex views, you may need to increase the value even more. In SQL*Plus, the COLUMN command specifies to wordwrap the results. If you tend to use long lines in your CREATE VIEW statements, you may need this to avoid having words chopped in the middle when line breaks occur.

Managing Synonyms

A synonym is an alternate name for an object. Primarily, synonyms enable multiple users to reference an object without them having to prefix the object name with the schema name. You manage synonyms by using the CREATE SYNONYM and DROP SYNONYM commands. You can also use Schema Manager to create and drop them.

Understanding synonyms

To better understand the issue that synonyms address, consider the AQUATIC_ANIMAL table in the SEAPARK schema. If you log on as the user named HAROLD and you want to select from that table, you could refer to it as SEAPARK.AQUATIC_ANIMAL. That's a lot to type. Further, if you were writing an application, you would need to embed the schema name throughout the application, forcing your application to depend on the tables stored in a schema named SEAPARK.

Synonyms provide you with a way to refer to objects in a schema-independent manner, as shown in Figure 15-2.

Once you create a synonym for an object, you can use that synonym in place of the object's name in SELECT statements, and in DML statements such as INSERT, UPDATE, or DELETE. With respect to Figure 15-2, you can use AQUATIC_ANIMAL any time that you want to refer to the SEAPARK.AQUATIC_ANIMAL table.

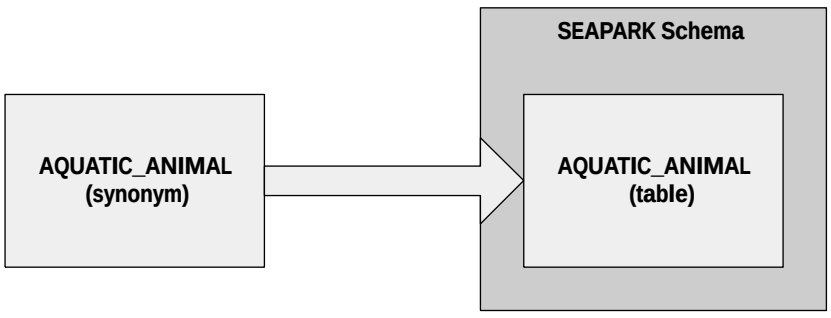


Figure 15-2: Synonyms point to schema objects.

Oracle uses synonyms extensively to refer to the data dictionary views, all of which are owned by the user named SYS. It's only because of these synonyms that you can select from DBA_TABLES without having to reference it as SYS.DBA_TABLES.

You can make a synonym for a table, a view, or even for another synonym. In addition, you can create synonyms for functions, packages, procedures, sequences, and database links to remote databases. After you create a synonym, you can use it as though it were the underlying object. Synonyms don't contain their own data; they are only a pointer to an object.

Types of Synonyms

Oracle supports two types of synonyms: private and public. *Private synonyms* are created by a user, and they apply only to statements issued by that one user. *Public synonyms* are created by the DBA and are shared by all users. Figure 15-3 illustrates the difference.

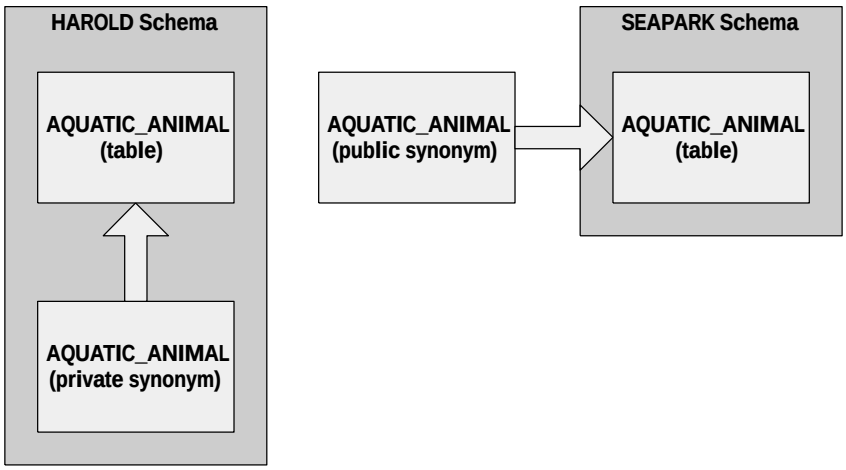


Figure 15-3: Public versus private synonyms

In Figure 15-3, you can see that a public synonym named AQUATIC_ANIMALS exists, and that it points to the SEAPARK.AQUATIC_ANIMALS table. All users but one will use this synonym. That one user is HAROLD. User HAROLD has his own private synonym pointing to his own copy of the AQUATIC_ANIMALS table.

Synonym names don't need to match table names, but making them match is a common practice.

Private synonyms always override public synonyms. When a SQL command contains an object name that isn't prefixed with a schema name, Oracle resolves the name by checking for objects in this order:

1. Objects and private synonyms owned by the current user
2. Public synonyms

Public synonyms are most often used when the purpose is to avoid having to code schema names within applications.

When to Use Synonyms

Application developers typically use private synonyms as shorthand for tables they need to reference often during SQL code development. This approach avoids the retyping of long table names. Public synonyms simplify the migration of applications. If the schema names are different but the tables are the same, synonyms allow an application to be migrated easily from one database to another.

Sometimes, you use public synonyms for common lookup tables that all database users need. The synonyms for the `DBA` views in the data dictionary provide a good example of this use.

Another use for synonyms is to enable two applications to refer to a single table with two different names. Views can also perform this task, but if the sole purpose is to give each application a business-rule appropriate name, a synonym may be a good solution. For example, an accounting department application may refer to the customer table as `CUSTOMER`, while a sales department application may use `CLIENT` instead. Although this wouldn't be an ideal situation to create, sometimes you do have to deal with legacy applications.

When creating public synonyms, Oracle users often mistakenly assume they can immediately share their tables with other Oracle users. Creating a public synonym for a table doesn't automatically enable other users to view or change the data in that table. You must still assign the appropriate privileges using Security Manager or the SQL `GRANT` command.

Creating synonyms

You can use the `CREATE SYNONYM` command to create synonyms, or you can use Enterprise Manager's Schema Manager. You will need the `CREATE SYNONYM` privilege, and if you are creating public synonyms, you will need the `CREATE PUBLIC SYNONYM` privilege.

Using SQL to Create Synonyms

The CREATE SYNONYM statement allows you to create both public and private synonyms. The syntax is extremely simple. The most difficult part is getting your fingers to type *synonym* correctly. To create a private synonym referring to a table in another schema, issue the following command:

```
CREATE SYNONYM animal FOR seapark.aquatic_animal;
```

Having created this synonym, you can select from ANIMAL instead of SEAPARK.AQUATIC_ANIMAL. Consider this example:

```
SQL> SELECT id_no, animal_name
  2  FROM animal;

  ID_NO ANIMAL_NAME
--------- --------------------
  100 Flipper
  105 Skipper
  112 Bopper
  151 Batty
```

Remember that creating a synonym doesn't give you access to another user's table. That user must grant you SELECT privileges for you to be able to select data from the table.

To create a public synonym on a table, add the PUBLIC keyword to the CREATE SYNONYM command. All database users may use the synonym you can create in this example:

```
CREATE PUBLIC SYNONYM animal FOR seapark.aquatic_animal;
```

Synonyms are also useful in making access to data in remote databases transparent to both users and applications. For example, if you have a remote database containing a human resource application, you might create a synonym like the following so that you could tie into it:

```
CREATE PUBLIC SYNONYM employee
FOR payroll.employee_public_data@hr;
```

With this synonym in place, your applications would be able to select from EMPLOYEE without having to worry about the fact that the table was actually in another database.

Using Schema Manager to Create Synonyms

You can use Schema Manager to create a synonym by following these steps:

1. Start Schema Manager and log on to the database.

2. Right-click the Synonyms folder and select Create from the pop-up menu.

   The General tab of the Create Synonym dialog box, shown in Figure 15-4, will display.

3. Fill in the fields of the General tab with the information needed to create the synonym.

   Figure 15-4 shows the `ANIMAL` synonym being created.

4. Click the Create button.

After you carry out these four easy steps, you have your synonym.

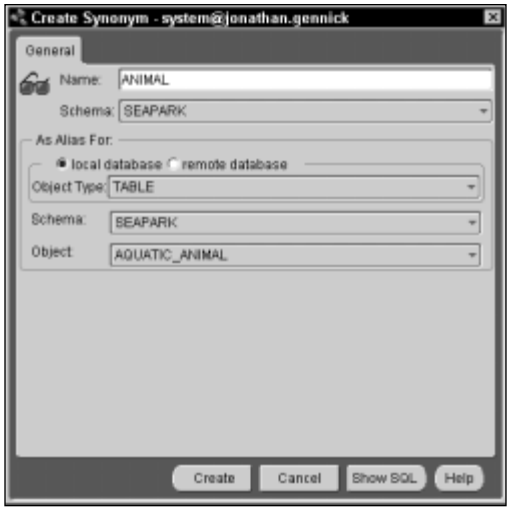


Figure 15-4: Creating the ANIMAL synonym

Removing synonyms

You can remove a synonym from the database by issuing a SQL statement or by using Schema Manager. The SQL statement to remove a synonym is as follows:

```
DROP [PUBLIC] SYNONYM synonym_name;
```

Be sure to include the keyword `PUBLIC` if you are dropping a public synonym. To use Schema Manager to remove a synonym, navigate to the synonym that you want to remove, right-click its name, and select Remove from the pop-up menu.

Using synonyms with the data dictionary

Only one data dictionary view is directly related to synonyms, and that's the DBA_SYNONYMS view. Like other DBA views, DBA_SYNONYMS is usually visible only to the DBA. If you aren't the DBA, use ALL_SYNONYMS instead.

Listing Synonyms Owned by a User

You can use the following query to list the synonyms owned by a particular user:

```
SELECT synonym_name, table_owner, table_name
FROM dba_synonyms
WHERE owner = 'username
```

In spite of the columns being named TABLE_OWNER and TABLE_NAME, this view does return information about synonyms pointing to views as well as to tables.

Listing Public Synonyms

To list the public synonyms that have been created on a database, query the DBA_SYNONYMS view using PUBLIC as the owner name. Consider this example:

```
SELECT synonym_name, table_owner, table_name
FROM dba_synonyms
WHERE owner = 'PUBLIC';
```

Oracle allows all users to see all public synonyms, regardless of whether they have access to the objects pointed to by those synonyms.

Listing Synonyms That Refer to an Object

Occasionally, you might need to find out if any synonyms refer to an object. You can do that by using the following query:

```
SELECT owner, synonym_name
FROM dba_synonyms
WHERE table_owner = 'object_owner'
AND table_name = 'object_name';
```

Similarly, you can also find all synonyms referring to objects owned by a user. Just drop the last condition of the WHERE clause.

Listing Synonyms That Reference a User

If you frequently export users from one database and import them into another, you'll soon find that when you export just one user, the Export utility doesn't export public synonyms that reference objects owned by that user. This makes sense, but it can be inconvenient if you are using the Export utility to migrate a

schema from a development database to a test database, because in that case, you probably do want the relevant public synonyms to go along.

If you need to re-create a user's public synonyms in another database, you can use the data dictionary to generate the necessary SQL commands automatically. Listing 15-2 shows an example of a SQL*Plus script that does this. It generates a file of `CREATE PUBLIC SYNONYM` commands for all public synonyms referencing objects owned by `SEAPARK`.

Listing 15-2: **Rebuilding public synonyms**

```
set linesize 200
set pagesize 0
set trimspool on
spool c:\a\seapark_syn.sql
select 'create public synonym '
       || synonym_name
       || ' for '
       || table_owner || '.' || table_name
       || DECODE (db_link, null, '','@' || db_link)
       || ';'
  from dba_synonyms
 where table_owner = 'SEAPARK'
   and owner = 'PUBLIC';
spool off
```

After executing these commands against your source database, the file seapark_syn.sql will contain `CREATE PUBLIC SYNONYM` commands for objects owned by `SEAPARK`. You can create those on the target database by logging on to that database with SQL*Plus and executing the file.

Managing Sequences

A sequence is an Oracle object used to deliver a series of unique numbers. Sequences are transaction-independent. Each time you access a sequence, the value of that sequence is incremented (or decremented) by a predetermined amount. Committing or rolling back a transaction doesn't affect changes to a sequence. Sequences are stored in the database. They don't occupy a lot of space like tables do, but their values persist through shutdowns and startups. Sequences are most often used to generate unique, primary keys for tables. The example shown in Listing 15-3 demonstrates how you can select a series of incrementing values from a sequence.

Listing 15-3: **Sequences are used to generate a series of sequential values.**

```
SQL> SELECT checkup_history_seq.NEXTVAL
  2  FROM dual;

 NEXTVAL
---------
        1

1 row selected.

SQL> SELECT checkup_history_seq.NEXTVAL
  2  FROM dual;

 NEXTVAL
---------
        2

1 row selected.

SQL> SELECT checkup_history_seq.NEXTVAL
  2  FROM dual;

 NEXTVAL
---------
        3

1 row selected.
```

By themselves, Oracle sequences aren't worth much. You have to write some code somewhere that uses the sequences that you create. Many databases, such as Microsoft SQL Server and Microsoft Access, allow you to define fields in a table that autoincrement each time a row is inserted. These autoincrement fields are most often used as primary key generators.

You can perform the same function from Oracle, but with Oracle, the incrementing function (the sequence) is separate from the field being incremented. You need to tie the two together with some code, and you'll see how to do that later in the section "Creating an Autoincrementing Field."

Determining when to use sequences

Consider using sequences whenever you are faced with one of the following situations:

- ✦ You want the primary key of a table to be a number, and you want that number to autoincrement each time you insert a new row into the table.

- You are creating an audit trail table, and you need to know the exact order in which the audit trail entries were made. Timestamps alone aren't usually enough to do this.

In general, consider using a sequence whenever you need to generate unique numbers.

Creating a sequence

You can use the `CREATE SEQUENCE` statement to create a sequence. Issue the statement using SQL*Plus, or you can use Schema Manager's GUI interface, and Schema Manager will write and execute the statement for you.

Before you create a sequence, consider the following questions:

- Do you want the sequence to start with 1, or do you want some other value used as the starting point?
- Do you want the values to increment by 1, or by some other value? Do you want the values to decrement?
- What do you want the behavior to be when the sequence reaches its maximum value?
- Must you have the values generated sequentially, or can you receive the values out of order?
- How many values can you afford to lose if the instance goes down?

When you create a sequence, Oracle allows you to choose the starting value. You may also set a maximum value, and you may set an increment value. The increment defaults to 1, but you may choose another increment if you prefer. You can create a descending sequence by making the increment a negative value. An increment of -1, for example, results in a sequence that counts down like this: 10, 9, 8, and so on.

You may specify maximum and minimum values when you create a sequence. The default minimum is 1. The default maximum is 1.0E+27 (1 with 27 zeros after it). You can also control Oracle's behavior once that maximum is hit. By default, Oracle will stop generating values when the sequence reaches its maximum. Using the `CYCLE` option, you can create sequences that automatically start over again at the beginning.

If you are using sequences to generate unique keys for records, you should avoid the `CYCLE` option, unless, that is, you have some way to be sure that older records have been deleted from the table by the time the sequence cycles back to the beginning again.

Another consideration when creating a sequence is whether you want the values to be generated in numeric order. This may seem like a silly issue. After all, why call it a sequence if the values aren't in sequence, right? Ordering is an issue only when you're using Parallel Server, and it's an issue because of how Oracle caches sequence

values in memory. Each instance accessing a database maintains a cache of sequence values in memory. This is to avoid the overhead of disk access each time a new sequence value is requested. You end up with the situation shown in Figure 15-5.

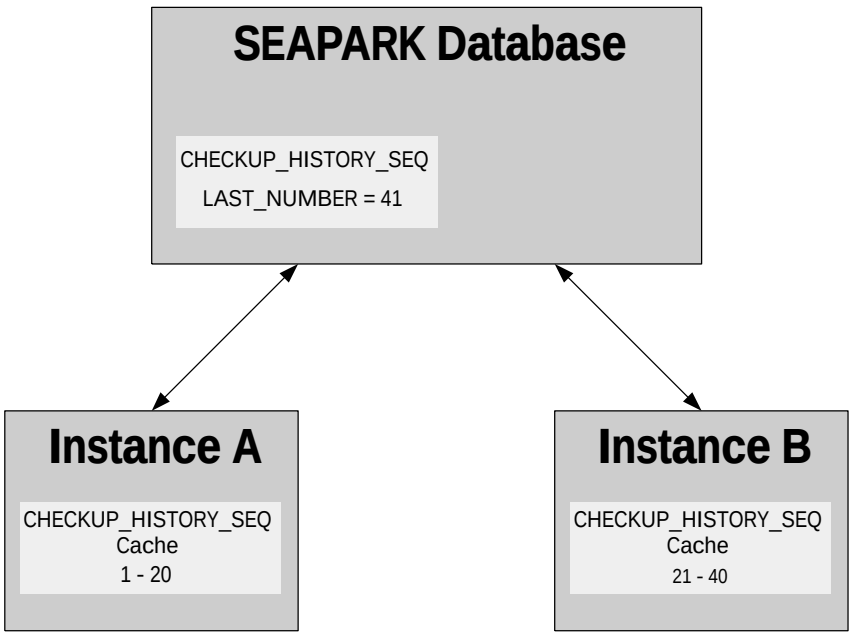


Figure 15-5: Each Oracle instance gets its own cache of sequence values.

In Figure 15-5, if two users are connected to instances A and B, respectively, and are alternately inserting new records into the database, the sequence values used will be 1, 21, 2, 22, 3, 23, and so forth. This really shouldn't present a problem if you are using the sequence only to generate unique keys. If you absolutely must have sequence values generated in order, you can use the `ORDER` option when creating the sequence. However, there's a slight performance cost in doing this.

Challenge anyone who claims that ordering is a requirement. Sometimes, in the case of an audit trail table, it really is a requirement. More often, it's an issue of someone's sense of "neatness" being violated by out-of-order values. If all you need is a unique number to identify a record, does it really matter if that number is in sequence or not?

The last issue to think about is how many sequence values you want to cache in memory. This is a performance tradeoff. By default, Oracle caches 20 values from each sequence in memory. That means that you can access a sequence 19 times without causing disk access. On the 20th time, Oracle will need to access the disk to refresh the cache. The tradeoff that you make when you cache sequence numbers is that the values in memory are lost if an instance crashes. This results in gaps. If

you are using sequences to generate primary keys, you could end up jum-ping from 100 to 121 and not have any records for numbers 101 to 120. You can use the NOCACHE option to prevent Oracle from caching values.

Using the NOCACHE option will hurt performance because Oracle will be forced to access the disk each time a new sequence value is requested. Also, by itself, NOCACHE won't eliminate the problem of gaps in autoincremented fields. There is still a risk of loss due to transactions that are rolled back, including uncommitted transactions that are rolled back due to an instance failure.

Using the CREATE SEQUENCE Statement

You can use the CREAT SEQUENCE statement to create a sequence from SQL*Plus. To use the command in its simplest form, taking all the defaults, you just need to supply a sequence name after the command. Consider this example:

```
SQL> CREATE SEQUENCE aquatic_animal_seq;

Sequence created.
```

If you're using a sequence for a primary key or for an audit trail log, then you may want to specify some nondefault settings. The command in the following example creates a sequence that would be useful for an audit trail log:

```
SQL> CREATE SEQUENCE aquatic_animal_audit_trail_seq
  2  START WITH 1
  3  INCREMENT BY 1
  4  ORDER
  5  NOCYCLE;

Sequence created.
```

The ORDER and NOCYCLE options are significant here. For an audit trail, you need to know the exact order in which events occur. Hence, you use the ORDER option to guarantee that each sequence value will be greater than the previous value. You use the NOCYCLE option to prevent the sequence from automatically wrapping around to 1 again, which could cause primary key violations if the audit trail table is keyed by the sequence value.

Using Schema Manager to Create a Sequence

You can use Schema Manager to create a sequence in much the same way that you create a table or any other object. Follow these steps:

1. Start Schema Manager and log on to the database.
2. Right-click on Sequences and select Create from the pop-up menu.
3. Fill in the General tab in the Create Sequence dialog box.

   Figure 15-6 shows the CHECKUP_HISTORY_SEQ sequence being created.

4. Click the Create button, and Schema Manager will execute the `CREATE SEQUENCE` command for you.

Figure 15-6: Using Schema Manager to create the CHECKUP_HISTORY_SEQ sequence

Creating an autoincrementing field

People moving to an Oracle environment after having worked with SQL Server or Microsoft Access often become frustrated with the task of defining an autoincrementing field in a table. While some database products make this easy by supporting autoincrementing fields as a distinct datatype, Oracle requires you to write code.

To create an autoincrementing field in Oracle, follow these steps:

1. Define the column that you want to increment as a number.
2. Create a sequence in support of that column.
3. Write a trigger to set the value of the column from the sequence whenever a row is inserted.
4. Optionally, write a trigger to prevent updates to the column.

As an example, let's say that you want the primary key for the `CHECKUP_HISTORY` table to automatically increment by 1 each time a new record is inserted. The structure of the `CHECKUP_HISTORY` table looks like this:

```
SQL> DESCRIBE checkup_history
```

```
Name                           Null?      Type
------------------------------ -------- -------------
CHECKUP_NO                     NOT NULL   NUMBER(10)
ID_NO                                     NUMBER(10)
CHECKUP_TYPE                              VARCHAR2(30)
CHECKUP_DATE                              DATE
DOCTOR_NAME                               VARCHAR2(50)
```

The CHECKUP_NO field is the primary key, and it is already a number. You can create a stored sequence for this field by using the following command:

```
CREATE SEQUENCE checkup_no_seq
NOCYCLE
MAXVALUE 9999999999
START WITH 2;
```

The maximum value was chosen to match the field definition. The CHECKUP_NO field is 10 digits long, so the maximum value that it can hold is 9,999,999,999. You could start the sequence at 1, but if the table already contains data, you might need to start at a higher number to avoid conflicts with the existing data. In this case, the sequence starts at 2.

Your next task is to make certain that the sequence is always used to set the CHECKUP_NO field each time a new row is inserted into the table. You could code all your application programs to do this, but a more robust approach is to create an insert trigger on the CHECKUP_HISTORY table. Listing 15-4 demonstrates a trigger that ensures that the key is always properly set, no matter how a row is inserted.

Listing 15-4: A trigger to implement an autoincrementing primary key field

```
CREATE OR REPLACE TRIGGER set_checkup_no
BEFORE INSERT ON checkup_history
FOR EACH ROW
DECLARE
  next_checkup_no NUMBER;
BEGIN
   --Get the next checkup number from the sequence.
   SELECT checkup_no_seq.NEXTVAL
   INTO next_checkup_no
   FROM dual;

   --Use the sequence number as the primary key
   --for the record being inserted.
   :new.checkup_no := next_checkup_no;
END;
/
```

With this trigger in place, the CHECKUP_NO field will always be set using the sequence that you have created. This will be true even if an INSERT statement specifies some other value for that field. The trigger will override any value supplied by the INSERT statement, ensuring that the sequence is always the source for primary keys.

In this situation, it's possible for someone to update an existing record and change the primary key value to something that might conflict with a future insertion. You may want to head off that possibility by creating an update trigger like the following:

```
CREATE OR REPLACE TRIGGER upd_checkup_no
BEFORE UPDATE OF checkup_no ON checkup_history
FOR EACH ROW
BEGIN
   RAISE_APPLICATION_ERROR (-20000,
      'UPD_CHECKUP_NO Trigger: Updates of the CHECKUP_NO field'
      || ' are not allowed.');
END;
/
```

The BEFORE UPDATE OF checkup_no clause ensures that this trigger will fire only when a user attempts to update the CHECKUP_NO column. This minimizes the overhead incurred by normal updates. If an illegal update is detected, the RAISE_APPLICATION_ERROR procedure is used to send a human readable error message back to the application.

Altering a sequence

You can alter a sequence in one of two ways. You can use the ALTER SEQUENCE command to change any of the options, such as the amount of values cached, the minimum and maximum values, and the increment. Or you can use Schema Manager by right-clicking on a sequence name and selecting Edit from the pop-up menu.

Unfortunately, you can't alter a current value for the sequence itself. The best way to change the value of a sequence is to drop it and re-create it using the START WITH clause. When you do this, you will lose any access granted on the sequence. You'll have to regrant access after you re-create it. You will also need to recompile any triggers that depend on the sequence.

If you absolutely must change the value of a sequence without dropping and recreating it, you may be able to use a workaround to temporarily change the increment value so that the next time a value is retrieved, the sequence will be advanced to where you want it to be. For this workaround to work reliably, you must be absolutely certain that no other users will access the sequence while you are making the change.

As an example, assume that you must change the CHECKUP_HISTORY_SEQ sequence so that the next value to be retrieved is 1,000. The first step is to ascertain the current value of the sequence. You can do that as shown in the following example:

```
SQL> SELECT checkup_history_seq.NEXTVAL
  2  FROM dual;

  NEXTVAL
---------
        2
```

The current value is 2. The next value to be selected is now 3. You want the next value to be 1,000. Subtract 3 from 1,000, and you have an increment of 997. Now, issue the commands shown in Listing 15-5 to temporarily set the increment to 997, select from the sequence, and then reset the increment back to 1.

Listing 15-5: **Changing the sequence value**

```
SQL> ALTER SEQUENCE checkup_history_seq
  2     INCREMENT BY 997;

Sequence altered.

SQL> SELECT checkup_history_seq.NEXTVAL
  2  FROM dual;

  NEXTVAL
---------
      999

1 row selected.

SQL> ALTER SEQUENCE checkup_history_seq
  2     INCREMENT BY 1;

Sequence altered.
```

You can see that after the increment was modified, the sequence returned a value of 2+997, or 999. Setting the increment back to 1 afterwards causes the next value retrieved to be 1,000, as shown in the following example:

```
SQL> SELECT checkup_history_seq.NEXTVAL
  2  FROM dual;

  NEXTVAL
---------
     1000
```

For this to work, you need to be certain that no other user accesses the sequence while you have the increment modified. Otherwise, you may find yourself jumping to 1997 (2 + 997 + 997 + 1) instead of 1,000.

Sometimes the risk of other users accessing a sequence while you are modifying it is worth taking. You do have to be sure that you can stand a big gap in the values, though. While making a change on the fly, when other users could potentially access the sequence, write out all the necessary SQL statements ahead of time and execute them from a script. This greatly reduces the period of time during which you are exposed.

Removing a sequence

You can use the `DROP SEQUENCE` command to remove a sequence. Consider this example:

```
DROP SEQUENCE checkup_history_seq;
```

You may also use Schema Manager to drop sequences. Simply right-click the sequence that you want to drop and select Remove from the pop-up menu.

Dropping a sequence causes Oracle to remove any privileges granted on the sequence. Synonyms remain but are invalid. Triggers using the dropped sequence remain but will also be invalid, and you'll get error messages each time those triggers are fired.

Using sequences with the data dictionary

The `DBA_SEQUENCES` view returns information about all the sequences defined in a database. You are typically the only person who can see this view. If you aren't the DBA, use `ALL_SEQUENCES` or `USER_SEQUENCES` instead. To get a list of all sequences owned by a particular user, issue a query like the one that follows:

```
SELECT sequence_name
FROM dba_sequences
WHERE sequence_owner = 'schema_name';
```

To find out information about one sequence in particular, issue the following query:

```
SELECT *3
FROM dba_sequences
WHERE sequence_owner = 'schema_name'
AND sequence_name = 'sequence_name';
```

One of the values returned by the `DBA_SEQUENCES` view is named `LAST_NUMBER`. This isn't the last sequence number returned to a user. It is the most recent value recorded on disk. It is set when Oracle fills its cache with values for the sequence, and it will be one increment higher than the highest value that Oracle has cached so far.

Summary

In this chapter, you learned:

- ✦ Views help streamline access to the data by hiding complexities of the underlying tables. Views can also help to separate data for different users as a security measure. You can reference views almost anywhere you reference a table.
- ✦ Views are stored in the database as a SQL `SELECT` command and have no data of their own.
- ✦ Synonyms enable developers to reference tables with shorthand names. Synonyms also allow different applications to reference the same object with different names. Synonyms can help simplify migration of applications from one schema or database instance to another.
- ✦ Oracle supports two types of synonyms: public and private. Public synonyms affect all users in the database. Private synonyms affect only the user who owns them. Private synonyms take precedence over public synonyms of the same name.
- ✦ You can use sequences to generate unique numbers for primary keys. Sequences are safer than other application-based methods of key generation because they are independent of transactions.
- ✦ If you are creating a sequence in support of an audit trail table, you need to be sure that the sequence numbers are assigned in strict order. Be sure to specify the `ORDER` option.

✦ ✦ ✦