23
Managing Views, Sequences, and Synonyms

You can create and manage views, sequences, and synonyms with Oracle Database.

- Managing Views
  You can perform tasks such as creating views, replacing views, altering views, and dropping views.
- Managing Sequences
  You can perform tasks such as creating sequences, altering sequences, using sequences, and dropping sequences.
- Managing Synonyms
  You can perform tasks such as creating synonyms, using synonyms, and dropping synonyms.
- Views, Synonyms, and Sequences Data Dictionary Views
  You can query data dictionary views for information about views, synonyms, and sequences.

23.1 Managing Views

You can perform tasks such as creating views, replacing views, altering views, and dropping views.

> **Live SQL:**
>
> To view and run examples related to managing views on Oracle Live SQL, go to *Oracle Live SQL: Creating, Replacing, and Dropping a View*.

- About Views
  A **view** is a logical representation of a table or combination of tables. In essence, a view is a stored query.
- Creating Views and Join Views
  You can create views using the `CREATE VIEW` statement. Each view is defined by a query that references tables, materialized views, or other views. You can also create join views that specify multiple base tables or views in the `FROM` clause.
- Replacing Views
  You can replace a view by dropping it and re-creating it or by issuing a `CREATE VIEW` statement that contains the `OR REPLACE` clause.
- Using Views in Queries
  You can query a view. You can also perform data manipulation language (DML) operations on views, with some restrictions.
- DML Statements and Join Views
  Restrictions apply when issuing DML statements on join views.

- Altering Views
  You use the `ALTER VIEW` statement only to explicitly recompile a view that is invalid.
- Dropping Views
  You can drop a view with the `DROP VIEW` statement.

23.1.1 About Views

A **view** is a logical representation of a table or combination of tables. In essence, a view is a stored query.

A view derives its data from the tables on which it is based. These tables are called **base tables**. Base tables might in turn be actual tables or might be views themselves. All operations performed on a view actually affect the base table of the view. You can use views in almost the same way as tables. You can query, update, insert into, and delete from views, just as you can standard tables.

Views can provide a different representation (such as subsets or supersets) of the data that resides within other tables and views. Views are very powerful because they allow you to tailor the presentation of data to different types of users.

> **Note:**
>
> One special type of view is the editioning view, which is used only to support online upgrade of applications using edition-based redefinition. The remainder of this section on managing views describes all views except editioning views. See *Oracle Database Development Guide* for a discussion of editioning views and edition-based redefinition.

> **See Also:**
>
> *Oracle Database Concepts* for an overview of views

23.1.2 Creating Views and Join Views

You can create views using the `CREATE VIEW` statement. Each view is defined by a query that references tables, materialized views, or other views. You can also create join views that specify multiple base tables or views in the `FROM` clause.

- Creating Views
  You can create a view with the `CREATE VIEW` statement.
- Creating Join Views
  You can also create views that specify multiple base tables or views in the `FROM` clause of a `CREATE VIEW` statement. These are called **join views**.
- Expansion of Defining Queries at View Creation Time
  When a view is created, Oracle Database expands any wildcard (*) in a top-level view query into a column list. The resulting query is stored in the data dictionary; any subqueries are left intact.

- Creating Views with Errors
  If there are no syntax errors in a `CREATE VIEW` statement, then the database can create the view even if the defining query of the view cannot be executed. In this case, the view is considered "created with errors."

23.1.2.1 Creating Views

You can create a view with the `CREATE VIEW` statement.

To create a view, you must meet the following requirements:

- To create a view in your schema, you must have the `CREATE VIEW` privilege. To create a view in another user's schema, you must have the `CREATE ANY VIEW` system privilege. You can acquire these privileges explicitly or through a role.
- The owner of the view (whether it is you or another user) must have been explicitly granted privileges to access all objects referenced in the view definition. The owner *cannot* have obtained these privileges through roles. Also, the functionality of the view depends on the privileges of the view owner. For example, if the owner of the view has only the `INSERT` privilege for Scott's `emp` table, then the view can be used only to insert new rows into the `emp` table, not to `SELECT`, `UPDATE`, or `DELETE` rows.
- If the owner of the view intends to grant access to the view to other users, the owner must have received the object privileges to the base objects with the `GRANT OPTION` or the system privileges with the `ADMIN OPTION`.

You can create views using the `CREATE VIEW` statement. Each view is defined by a query that references tables, materialized views, or other views. As with all subqueries, the query that defines a view cannot contain the `FOR UPDATE` clause.

The following statement creates a view on a subset of data in the `hr.departments` table:

```
CREATE VIEW departments_hq AS
      SELECT department_id, department_name, location_id
      FROM hr.departments
      WHERE location_id = 1700
    WITH CHECK OPTION CONSTRAINT departments_hq_cnst;
```

The query that defines the `departments_hq` view references only rows in location 1700. Furthermore, the `CHECK OPTION` creates the view with the constraint (named `departments_hq_cnst`) so that `INSERT` and `UPDATE` statements issued against the view cannot result in rows that the view cannot select. For example, the following `INSERT` statement successfully inserts a row into the `departments` table with the `departments_hq` view, which contains all rows with location 1700:

```
INSERT INTO departments_hq VALUES (300, 'NETWORKING', 1700);
```

However, the following `INSERT` statement returns an error because it attempts to insert a row for location 2700, which cannot be selected using the `departments_hq` view:

```
INSERT INTO departments_hq VALUES (301, 'TRANSPORTATION', 2700);
```

The view could have been constructed specifying the `WITH READ ONLY` clause, which prevents any updates, inserts, or deletes from being done to the base table through the view. If no `WITH` clause is specified, the view, with some restrictions, is inherently updatable.

You can also create views with invisible columns. For example, the following statements creates the `departments_hq_man` view and makes the `manager_id` column invisible:

```
CREATE VIEW departments_hq_man
        (department_id, department_name, manager_id INVISIBLE, location_id)
     AS SELECT department_id, department_name, manager_id, location_id
     FROM hr.departments
     WHERE location_id = 1700
   WITH CHECK OPTION CONSTRAINT departments_hq_man_cnst;
```

See Also:

- *Oracle Database SQL Language Reference* for syntax and semantics of the `CREATE VIEW` statement
- "Understand Invisible Columns"

23.1.2.2 Creating Join Views

You can also create views that specify multiple base tables or views in the `FROM` clause of a `CREATE VIEW` statement. These are called **join views**.

The following statement creates the `division1_staff` view that joins data from the `emp` and `dept` tables:

```
CREATE VIEW division1_staff AS
     SELECT ename, empno, job, dname
     FROM emp, dept
     WHERE emp.deptno IN (10, 30)
        AND emp.deptno = dept.deptno;
```

An **updatable join view** is a join view where `UPDATE`, `INSERT`, and `DELETE` operations are allowed. See "Updating a Join View" for further discussion.

23.1.2.3 Expansion of Defining Queries at View Creation Time

When a view is created, Oracle Database expands any wildcard (*) in a top-level view query into a column list. The resulting query is stored in the data dictionary; any subqueries are left intact.

The column names in an expanded column list are enclosed in quotation marks to account for the possibility that the columns of the base object were originally entered with quotes and require them for the query to be syntactically correct.

As an example, assume that the `dept` view is created as follows:

```
CREATE VIEW dept AS SELECT * FROM scott.dept;
```

The database stores the defining query of the `dept` view as:

```
SELECT "DEPTNO", "DNAME", "LOC" FROM scott.dept;
```

Views created with errors do not have wildcards expanded. However, if the view is eventually compiled without errors, wildcards in the defining query are expanded.

23.1.2.4 Creating Views with Errors

If there are no syntax errors in a `CREATE VIEW` statement, then the database can create the view even if the defining query of the view cannot be executed. In this case, the view is considered "created with errors."

For example, when a view is created that refers to a nonexistent table or an invalid column of an existing table, or when the view owner does not have the required privileges, the view can be created anyway and entered into the data dictionary. However, the view is not yet usable.

To create a view with errors, you must include the `FORCE` clause of the `CREATE VIEW` statement.

```
CREATE FORCE VIEW AS ...;
```

By default, views with errors are created as `INVALID`. When you try to create such a view, the database returns a message indicating the view was created with errors. If conditions later change so that the query of an invalid view can be executed, the view can be recompiled and be made valid (usable). For information changing conditions and their impact on views, see "Managing Object Dependencies".

23.1.3 Replacing Views

You can replace a view by dropping it and re-creating it or by issuing a `CREATE VIEW` statement that contains the `OR REPLACE` clause.

To replace a view, you must have all of the privileges required to drop and create a view. If the definition of a view must change, the view must be replaced; you cannot use an `ALTER VIEW` statement to change the definition of a view. You can replace views in the following ways:

- You can drop and re-create the view.

  > **Note:**
  >
  > When a view is dropped, all grants of corresponding object privileges are revoked from roles and users. After the view is re-created, privileges must be regranted.

- You can redefine the view with a `CREATE VIEW` statement that contains the `OR REPLACE` clause. The `OR REPLACE` clause replaces the current definition of a view and preserves the current security authorizations. For example, assume that you created the `sales_staff` view as shown earlier, and, in addition, you granted several object privileges to roles and other users. However, now you must redefine the `sales_staff` view to change the department number specified in the `WHERE` clause. You can replace the current version of the `sales_staff` view with the following statement:

  ```
  CREATE OR REPLACE VIEW sales_staff AS
       SELECT empno, ename, deptno
       FROM emp
       WHERE deptno = 30
       WITH CHECK OPTION CONSTRAINT sales_staff_cnst;
  ```

Before replacing a view, consider the following effects:

- Replacing a view replaces the view definition in the data dictionary. All underlying objects referenced by the view are not affected.

- If a constraint in the `CHECK OPTION` was previously defined but not included in the new view definition, the constraint is dropped.
- All views dependent on a replaced view become invalid (not usable). In addition, dependent PL/SQL program units may become invalid, depending on what was changed in the new version of the view. For example, if only the `WHERE` clause of the view changes, dependent PL/SQL program units remain valid. However, if any changes are made to the number of view columns or to the view column names or data types, dependent PL/SQL program units are invalidated. See "Managing Object Dependencies" for more information on how the database manages such dependencies.

23.1.4 Using Views in Queries

You can query a view. You can also perform data manipulation language (DML) operations on views, with some restrictions.

To issue a query or an `INSERT`, `UPDATE`, or `DELETE` statement against a view, you must have the `SELECT`, `READ`, `INSERT`, `UPDATE`, or `DELETE` object privilege for the view, respectively, either explicitly or through a role.

Views can be queried in the same manner as tables. For example, to query the `Division1_staff` view, enter a valid `SELECT` statement that references the view:

```
SELECT * FROM Division1_staff;

ENAME        EMPNO       JOB             DNAME
-----------------------------------------------------
CLARK         7782       MANAGER         ACCOUNTING
KING          7839       PRESIDENT       ACCOUNTING
MILLER        7934       CLERK           ACCOUNTING
ALLEN         7499       SALESMAN        SALES
WARD          7521       SALESMAN        SALES
JAMES         7900       CLERK           SALES
TURNER        7844       SALESMAN        SALES
MARTIN        7654       SALESMAN        SALES
BLAKE         7698       MANAGER         SALES
```

With some restrictions, rows can be inserted into, updated in, or deleted from a base table using a view. The following statement inserts a new row into the `emp` table using the `sales_staff` view:

```
INSERT INTO sales_staff
    VALUES (7954, 'OSTER', 30);
```

Restrictions on DML operations for views use the following criteria in the order listed:

1. If a view is defined by a query that contains `SET` or `DISTINCT` operators, a `GROUP BY` clause, or a group function, then rows cannot be inserted into, updated in, or deleted from the base tables using the view.
2. If a view is defined with `WITH CHECK OPTION`, a row cannot be inserted into, or updated in, the base table (using the view), if the view cannot select the row from the base table.
3. If a `NOT NULL` column that does not have a `DEFAULT` clause is omitted from the view, then a row cannot be inserted into the base table using the view.
4. If the view was created by using an expression, such as `DECODE(deptno, 10, "SALES", ...)`, then rows cannot be inserted into or updated in the base table using the view.

The constraint created by `WITH CHECK OPTION` of the `sales_staff` view only allows rows that have a department number of 30 to be inserted into, or updated in, the `emp` table. Alternatively, assume that the `sales_staff` view is defined by the following statement (that is, excluding the `deptno` column):

```
CREATE VIEW sales_staff AS
    SELECT empno, ename
    FROM emp
    WHERE deptno = 10
    WITH CHECK OPTION CONSTRAINT sales_staff_cnst;
```

Considering this view definition, you can update the `empno` or `ename` fields of existing records, but you cannot insert rows into the `emp` table through the `sales_staff` view because the view does not let you alter the `deptno` field. If you had defined a `DEFAULT` value of 10 on the `deptno` field, then you could perform inserts.

When a user attempts to reference an invalid view, the database returns an error message to the user:

```
ORA-04063: view 'view_name' has errors
```

This error message is returned when a view exists but is unusable due to errors in its query (whether it had errors when originally created or it was created successfully but became unusable later because underlying objects were altered or dropped).

23.1.5 DML Statements and Join Views

Restrictions apply when issuing DML statements on join views.

- Updating a Join View
  An updatable join view (also referred to as a **modifiable join view**) is a view that contains multiple tables in the top-level `FROM` clause of the `SELECT` statement, and is not restricted by the `WITH READ ONLY` clause.
- Key-Preserved Tables
  A table is key-preserved if every key of the table can also be a key of the result of the join that is based on the table. So, a key-preserved table has its keys preserved through a join.
- Rules for DML Statements and Join Views
  The general rule is that any `UPDATE`, `DELETE`, or `INSERT` statement on a join view can modify only one underlying base table.
- Updating Views That Involve Outer Joins
  Views that involve outer joins are modifiable in some cases.
- Using the UPDATABLE_ COLUMNS Views
  A set of views can assist you in identifying inherently updatable join views.

23.1.5.1 Updating a Join View

An updatable join view (also referred to as a **modifiable join view**) is a view that contains multiple tables in the top-level `FROM` clause of the `SELECT` statement, and is not restricted by the `WITH READ ONLY` clause.

The rules for updatable join views are shown in the following table. Views that meet these criteria are said to be inherently updatable.

Rule	Description
General Rule	Any `INSERT`, `UPDATE`, or `DELETE` operation on a join view can modify only one underlying base table at a time.
`UPDATE` Rule	Rows from a join view can be updated if the join column keys in the base tables are unique. That is, the `WHERE` clause in the `UPDATE` statement must be deterministic. If the base tables are not key-preserved, you must ensure that the join column keys are unique. If the view is defined with the `WITH CHECK OPTION` clause, then all join columns and all columns of repeated tables are not updatable.
`DELETE` Rule	Rows from a join view can be deleted as long as there is exactly one key-preserved table in the join. The key preserved table can be repeated in the `FROM` clause. If the view is defined with the `WITH CHECK OPTION` clause and the key preserved table is repeated, then the rows cannot be deleted from the view.
`INSERT` Rule	An `INSERT` statement must not explicitly or implicitly refer to the columns of a **non-key-preserved table**. If the join view is defined with the `WITH CHECK OPTION` clause, `INSERT` statements are not permitted.

There are data dictionary views that indicate whether the columns in a join view are inherently updatable. See "Using the UPDATABLE_ COLUMNS Views" for descriptions of these views.

> **Note:**
>
> There are some additional restrictions and conditions that can affect whether a join view is inherently updatable. Specifics are listed in the description of the `CREATE VIEW` statement in the *Oracle Database SQL Language Reference.*
>
> If a view is not inherently updatable, it can be made updatable by creating an `INSTEAD OF` trigger on it. See *Oracle Database PL/SQL Language Reference* for information about triggers.
>
> Additionally, if a view is a join on other nested views, then the other nested views must be mergeable into the top level view. For a discussion of mergeable and unmergeable views, and more generally, how the optimizer optimizes statements that reference views, see the *Oracle Database SQL Tuning Guide.*

Examples illustrating the rules for inherently updatable join views, and a discussion of key-preserved tables, are presented in following sections. The examples in these sections work only if you explicitly define the primary and foreign keys in the tables, or define unique indexes. The following statements create the appropriately constrained table definitions for `emp` and `dept`.

```
CREATE TABLE dept (
      deptno        NUMBER(4) PRIMARY KEY,
      dname         VARCHAR2(14),
      loc           VARCHAR2(13));

CREATE TABLE emp (
      empno        NUMBER(4) PRIMARY KEY,
      ename        VARCHAR2(10),
      job          VARCHAR2(9),
      mgr          NUMBER(4),
      sal          NUMBER(7,2),
```

```
      comm         NUMBER(7,2),
      deptno       NUMBER(2),
      FOREIGN KEY (DEPTNO) REFERENCES DEPT(DEPTNO));
```

You could also omit the primary and foreign key constraints listed in the preceding example, and create a `UNIQUE INDEX` on `dept (deptno)` to make the following examples work.

The following statement created the `emp_dept` join view which is referenced in the examples:

```
CREATE VIEW emp_dept AS
      SELECT emp.empno, emp.ename, emp.deptno, emp.sal, dept.dname, dept.loc
      FROM emp, dept
      WHERE emp.deptno = dept.deptno
         AND dept.loc IN ('DALLAS', 'NEW YORK', 'BOSTON');
```

23.1.5.2 Key-Preserved Tables

A table is key-preserved if every key of the table can also be a key of the result of the join that is based on the table. So, a key-preserved table has its keys preserved through a join.

> **Note:**
>
> It is not necessary that the key or keys of a table be selected for it to be key preserved. It is sufficient that if the key or keys were selected, then they would also be keys of the result of the join.

The concept of a key-preserved table is fundamental to understanding the restrictions on modifying join views. Each row in a key-preserved table appears at most only once in a join view based on this table. If a table `T` is joined to a table `S` using the condition `T.col1 = S.col3`, then `T` is key-preserved if the join key is `S.col3` are unique. The data in a table is not relevant when determining whether a table is key-preserved. Instead, the constraints on the table determine if a table is key-preserved. Key-preserved tables are joined with a source table using the primary key or unique key of the source table.

For example, in the `emp_dept` view, because `emp` is joined with the primary key of `dept`, `emp` is a key-preserved table. If, in the `emp` table, there was at most one employee in each department, then `deptno` would be unique in the result of a join of `emp` and `dept`, but `dept` would still not be a key-preserved table.

If you select all rows from the `emp_dept` view, the results are:

```
EMPNO      ENAME      DEPTNO  DNAME          LOC
---------- ---------- ------- -------------- -----------
      7782 CLARK           10 ACCOUNTING     NEW YORK
      7839 KING            10 ACCOUNTING     NEW YORK
      7934 MILLER          10 ACCOUNTING     NEW YORK
      7369 SMITH           20 RESEARCH       DALLAS
      7876 ADAMS           20 RESEARCH       DALLAS
      7902 FORD            20 RESEARCH       DALLAS
      7788 SCOTT           20 RESEARCH       DALLAS
      7566 JONES           20 RESEARCH       DALLAS
8 rows selected.
```

In this view, `emp` is a key-preserved table, because `empno` is a key of the `emp` table, and also a key of the result of the join. `dept` is *not* a key-preserved table, because although `deptno` is a key of the `dept` table, it is not a key of the join.

23.1.5.3 Rules for DML Statements and Join Views

The general rule is that any UPDATE, DELETE, or INSERT statement on a join view can modify only one underlying base table.

- UPDATE Statements and Join Views
  Examples illustrate UPDATE statements that can modify join views.
- DELETE Statements and Join Views
  For most join views, a delete is successful only if there is *one and only one* key-preserved table in the join. The key-preserved table can be repeated in the FROM clause.
- INSERT Statements and Join Views
  Examples illustrate INSERT statements that can modify join views.

23.1.5.3.1 UPDATE Statements and Join Views

Examples illustrate UPDATE statements that can modify join views.

Starting with Oracle Database Release 21c, it is not mandatory for all updatable columns in a join view to map to columns of a key-preserved table. The columns in a non-key-preserved table can be updated if the UPDATE operation only updates columns from a single table and the update is deterministic, meaning that it updates each row only once.

The following example shows an UPDATE statement that successfully modifies the emp_dept view:

```
UPDATE emp_dept
     SET    sal = sal * 1.10
     WHERE  deptno = 10;
```

The following UPDATE statement successfully modifies the LOC column in the DEPT table (the non-key-preserved table) because it updates only one row in the EMP table:

```
UPDATE emp_dept
     SET    loc = 'BOSTON'
     WHERE  ename = 'SMITH';
```

The following UPDATE statement results in an ORA-30926 error because the update operation is non-deterministic:

```
UPDATE emp_dept
     SET    loc = 'BOSTON'
     WHERE  ename = 'S%';
```

A row from the dept table is joined to multiple rows from emp table (with ename = 'SCOTT' and ename = 'SMITH'). Therefore, an attempt is made to modify the same row multiple times. To make the UPDATE is deterministic, ensure that a row from the dept table is only joined to one row from emp table.

In general, all updatable columns of a join view must map to columns of a key-preserved table. If the view is defined using the WITH CHECK OPTION clause, then all join columns and all columns taken from tables that are referenced more than once in the view are not modifiable.

So, for example, if the emp_dept view were defined using WITH CHECK OPTION, the following UPDATE statement would fail:

```
UPDATE emp_dept
     SET    deptno = 10
     WHERE  ename = 'SMITH';
```

The statement fails because it is trying to update a join column.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for syntax and additional information about the `UPDATE` statement

23.1.5.3.2 DELETE Statements and Join Views

For most join views, a delete is successful only if there is *one and only one* key-preserved table in the join. The key-preserved table can be repeated in the `FROM` clause.

The following `DELETE` statement works on the `emp_dept` view:

```
DELETE FROM emp_dept
     WHERE ename = 'SMITH';
```

This `DELETE` statement on the `emp_dept` view is valid because it can be translated to a `DELETE` operation on the base `emp` table, and because the `emp` table is the only key-preserved table in the join.

In the following view, a `DELETE` operation is permitted, because although there are two key-preserved tables, they are the same table. That is, the key-preserved table is repeated. In this case, the delete statement operates on the first table in the `FROM` clause (`e1`, in this example):

```
CREATE VIEW emp_emp AS
     SELECT e1.ename, e2.empno, e2.deptno
     FROM emp e1, emp e2
     WHERE e1.empno = e2.empno;
```

If a view is defined using the `WITH CHECK OPTION` clause and the key-preserved table is repeated, rows cannot be deleted from such a view.

```
CREATE VIEW emp_mgr AS
     SELECT e1.ename, e2.ename mname
     FROM emp e1, emp e2
     WHERE e1.mgr = e2.empno
     WITH CHECK OPTION;
```

Note:

- If the DELETE statement uses the same column in its WHERE clause that was used to create the view as a join condition, then the delete operation can be successful when there are different key-preserved tables in the join. In this case, the DELETE statement operates on the first table in the FROM clause, and the tables in the FROM clause can be different from the tables in the WHERE clause.
- The DELETE statement is successful, even if it does not use the WHERE clause.
- The DELETE statement is successful, even if it uses a different column in its WHERE clause than the one that was used to create the view as a join condition.
- The DELETE statement operates on the second table in the FROM clause in all the cases, because no primary key is defined on the second table.
- If a primary key is defined on the second table, then the DELETE statement operates on the first table in the FROM clause.

See Also:

Oracle Database SQL Language Reference for syntax and additional information about the DELETE statement

23.1.5.3.3 INSERT Statements and Join Views

Examples illustrate INSERT statements that can modify join views.

The following INSERT statement on the emp_dept view succeeds:

```
INSERT INTO emp_dept (ename, empno, deptno)
   VALUES ('KURODA', 9010, 40);
```

This statement works because only one key-preserved base table is being modified (emp), and 40 is a valid deptno in the dept table (thus satisfying the FOREIGN KEY integrity constraint on the emp table).

An INSERT statement, such as the following, would fail for the same reason that such an UPDATE on the base emp table would fail: the FOREIGN KEY integrity constraint on the emp table is violated (because there is no deptno 77).

```
INSERT INTO emp_dept (ename, empno, deptno)
   VALUES ('KURODA', 9010, 77);
```

The following INSERT statement would fail with an error (ORA-01776 cannot modify more than one base table through a join view):

```
INSERT INTO emp_dept (empno, ename, loc)
   VALUES (9010, 'KURODA', 'BOSTON');
```

An INSERT cannot implicitly or explicitly refer to columns of a non-key-preserved table. If the join view is defined using the WITH CHECK OPTION clause, then you cannot perform an INSERT to it.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for syntax and additional information about the `INSERT` statement

23.1.5.4 Updating Views That Involve Outer Joins

Views that involve outer joins are modifiable in some cases.

For example:

```
CREATE VIEW emp_dept_oj1 AS
    SELECT empno, ename, e.deptno, dname, loc
    FROM emp e, dept d
    WHERE e.deptno = d.deptno (+);
```

The statement:

```
SELECT * FROM emp_dept_oj1;
```

Results in:

```
EMPNO   ENAME      DEPTNO  DNAME          LOC
------- ---------- ------- -------------- -------------
7369    SMITH      40      OPERATIONS     BOSTON
7499    ALLEN      30      SALES          CHICAGO
7566    JONES      20      RESEARCH       DALLAS
7654    MARTIN     30      SALES          CHICAGO
7698    BLAKE      30      SALES          CHICAGO
7782    CLARK      10      ACCOUNTING     NEW YORK
7788    SCOTT      20      RESEARCH       DALLAS
7839    KING       10      ACCOUNTING     NEW YORK
7844    TURNER     30      SALES          CHICAGO
7876    ADAMS      20      RESEARCH       DALLAS
7900    JAMES      30      SALES          CHICAGO
7902    FORD       20      RESEARCH       DALLAS
7934    MILLER     10      ACCOUNTING     NEW YORK
7521    WARD       30      SALES          CHICAGO
14 rows selected.
```

Columns in the base `emp` table of `emp_dept_oj1` are modifiable through the view, because `emp` is a key-preserved table in the join.

The following view also contains an outer join:

```
CREATE VIEW emp_dept_oj2 AS
SELECT e.empno, e.ename, e.deptno, d.dname, d.loc
FROM emp e, dept d
WHERE e.deptno (+) = d.deptno;
```

The following statement:

```
SELECT * FROM emp_dept_oj2;
```

Results in:

```
EMPNO      ENAME      DEPTNO    DNAME          LOC
---------- ---------- --------- -------------- ----
```

```
7782       CLARK      10        ACCOUNTING     NEW YORK
7839       KING       10        ACCOUNTING     NEW YORK
7934       MILLER     10        ACCOUNTING     NEW YORK
7369       SMITH      20        RESEARCH       DALLAS
7876       ADAMS      20        RESEARCH       DALLAS
7902       FORD       20        RESEARCH       DALLAS
7788       SCOTT      20        RESEARCH       DALLAS
7566       JONES      20        RESEARCH       DALLAS
7499       ALLEN      30        SALES          CHICAGO
7698       BLAKE      30        SALES          CHICAGO
7654       MARTIN     30        SALES          CHICAGO
7900       JAMES      30        SALES          CHICAGO
7844       TURNER     30        SALES          CHICAGO
7521       WARD       30        SALES          CHICAGO
                                 OPERATIONS     BOSTON
15 rows selected.
```

In this view, `emp` is no longer a key-preserved table, because the `empno` column in the result of the join can have nulls (the last row in the preceding `SELECT` statement). So, `UPDATE`, `DELETE`, and `INSERT` operations cannot be performed on this view.

In the case of views containing an outer join on other nested views, a table is key preserved if the view or views containing the table are merged into their outer views, all the way to the top. A view which is being outer-joined is currently merged only if it is "simple." For example:

```
SELECT col1, col2, ... FROM T;
```

The select list of the view has no expressions.

If you are in doubt whether a view is modifiable, then you can select from the `USER_UPDATABLE_COLUMNS` view to see if it is. For example:

```
SELECT owner, table_name, column_name, updatable FROM USER_UPDATABLE_COLUMNS
     WHERE TABLE_NAME = 'EMP_DEPT_VIEW';
```

This returns output similar to the following:

```
OWNER       TABLE_NAME      COLUMN_NAM      UPD
----------  ----------      ----------      ---
SCOTT       EMP_DEPT_V      EMPNO           NO
SCOTT       EMP_DEPT_V      ENAME           NO
SCOTT       EMP_DEPT_V      DEPTNO          NO
SCOTT       EMP_DEPT_V      DNAME           NO
SCOTT       EMP_DEPT_V      LOC             NO
5 rows selected.
```

23.1.5.5 Using the UPDATABLE_ COLUMNS Views

A set of views can assist you in identifying inherently updatable join views.

View	Description
`DBA_UPDATABLE_COLUMNS`	Shows all columns in all tables and views that are modifiable.
`ALL_UPDATABLE_COLUMNS`	Shows all columns in all tables and views accessible to the user that are modifiable.
`USER_UPDATABLE_COLUMNS`	Shows all columns in all tables and views in the user's schema that are modifiable.

The updatable columns in view `emp_dept` are shown below.

```
SELECT COLUMN_NAME, UPDATABLE
      FROM USER_UPDATABLE_COLUMNS
      WHERE TABLE_NAME = 'EMP_DEPT';

COLUMN_NAME                    UPD
------------------------------ ---
EMPNO                          YES
ENAME                          YES
DEPTNO                         YES
SAL                            YES
DNAME                          NO
LOC                            NO

6 rows selected.
```

> **See Also:**
>
> *Oracle Database Reference* for complete descriptions of the updatable column views

23.1.6 Altering Views

You use the `ALTER VIEW` statement only to explicitly recompile a view that is invalid.

To change the definition of a view, see "Replacing Views".

The `ALTER VIEW` statement lets you locate recompilation errors before run time. To ensure that the alteration does not affect the view or other objects that depend on it, you can explicitly recompile a view after altering one of its base tables.

To use the `ALTER VIEW` statement, the view must be in your schema, or you must have the `ALTER ANY TABLE` system privilege.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for syntax and additional information about the `ALTER VIEW` statement

23.1.7 Dropping Views

You can drop a view with the `DROP VIEW` statement.

You can drop any view contained in your schema. To drop a view in another user's schema, you must have the `DROP ANY VIEW` system privilege. Drop a view using the `DROP VIEW` statement. For example, the following statement drops the `emp_dept` view:

```
DROP VIEW emp_dept;
```

> **See Also:**
>
> *Oracle Database SQL Language Reference* for syntax and additional information about the `DROP VIEW` statement

23.2 Managing Sequences

You can perform tasks such as creating sequences, altering sequences, using sequences, and dropping sequences.

- About Sequences
  Sequences are database objects from which multiple users can generate unique integers. The sequence generator generates sequential numbers, which can be used to generate unique primary keys automatically, and to coordinate keys across multiple rows or tables.
- Creating Sequences
  Create a sequence using the `CREATE SEQUENCE` statement.
- Altering Sequences
  Alter a sequence using the `ALTER SEQUENCE` statement.
- Using Sequences
  A sequence can be accessed and incremented by multiple users.
- Dropping Sequences
  If a sequence is no longer required, you can drop the sequence using the `DROP SEQUENCE` statement.

23.2.1 About Sequences

Sequences are database objects from which multiple users can generate unique integers. The sequence generator generates sequential numbers, which can be used to generate unique primary keys automatically, and to coordinate keys across multiple rows or tables.

Without sequences, sequential values can only be produced programmatically. A new primary key value can be obtained by selecting the most recently produced value and incrementing it. This method requires a lock during the transaction and causes multiple users to wait for the next value of the primary key; this waiting is known as **serialization**. If developers have such constructs in applications, then you should encourage the developers to replace them with access to sequences. Sequences eliminate serialization and improve the concurrency of an application.

> **See Also:**
>
> *Oracle Database Concepts* for an overview of sequences

23.2.2 Creating Sequences

Create a sequence using the `CREATE SEQUENCE` statement.

To create a sequence in your schema, you must have the `CREATE SEQUENCE` system privilege. To create a sequence in another user's schema, you must have the `CREATE ANY SEQUENCE` privilege.

For example, the following statement creates a sequence used to generate employee numbers for the `empno` column of the `emp` table:

```
CREATE SEQUENCE emp_sequence
      INCREMENT BY 1
      START WITH 1
      NOMAXVALUE
      NOCYCLE
      CACHE 10;
```

Notice that several parameters can be specified to control the function of sequences. You can use these parameters to indicate whether the sequence is ascending or descending, the starting point of the sequence, the minimum and maximum values, and the interval between sequence values. The `NOCYCLE` option indicates that the sequence cannot generate more values after reaching its maximum or minimum value.

The `CACHE` clause preallocates a set of sequence numbers and keeps them in memory so that sequence numbers can be accessed faster. When the last of the sequence numbers in the cache has been used, the database reads another set of numbers into the cache.

The database might skip sequence numbers if you choose to cache a set of sequence numbers. For example, when an instance abnormally shuts down (for example, when an instance failure occurs or a `SHUTDOWN ABORT` statement is issued), sequence numbers that have been cached but not used are lost. Also, sequence numbers that have been used but not saved are lost as well. The database might also skip cached sequence numbers after an export and import. See *Oracle Database Utilities* for details.

See Also:

- *Oracle Database SQL Language Reference* for the `CREATE SEQUENCE` statement syntax
- *Oracle Real Application Clusters Administration and Deployment Guide* for information about using sequences in an Oracle Real Application Clusters environment

23.2.3 Altering Sequences

Alter a sequence using the `ALTER SEQUENCE` statement.

To alter a sequence, your schema must contain the sequence, you must have the `ALTER` object privilege on the sequence, or you must have the `ALTER ANY SEQUENCE` system privilege. You can alter a sequence to change any of the parameters that define how it generates sequence numbers. To change the starting point of a sequence, you can either drop the sequence and then re-create it, or use the `RESTART` clause to restart the sequence. For an ascending

sequence, the RESTART clause resets NEXTVAL to MINVALUE. For a descending sequence, NEXTVAL is reset to MAXVALUE.

The following example alters the emp_sequence sequence:

```
ALTER SEQUENCE emp_sequence
    INCREMENT BY 10
    MAXVALUE 10000
    CYCLE
    CACHE 20;
```

> **See Also:**
>
> *Oracle Database SQL Language Reference* for syntax and additional information about the ALTER SEQUENCE statement

23.2.4 Using Sequences

A sequence can be accessed and incremented by multiple users.

To use a sequence, your schema must contain the sequence or you must have been granted the SELECT object privilege for another user's sequence. Once a sequence is defined, it can be accessed and incremented by multiple users (who have SELECT object privilege for the sequence containing the sequence) with no waiting. The database does not wait for a transaction that has incremented a sequence to complete before that sequence can be incremented again.

The examples outlined in the following sections show how sequences can be used in parent/child table relationships. Assume an order entry system is partially comprised of two tables, orders_tab (parent table) and line_items_tab (child table), that hold information about customer orders. A sequence named order_seq is defined by the following statement:

```
CREATE SEQUENCE Order_seq
    START WITH 1
    INCREMENT BY 1
    NOMAXVALUE
    NOCYCLE
    CACHE 20;
```

- Referencing a Sequence
  A sequence is referenced in SQL statements with the NEXTVAL and CURRVAL pseudocolumns; each new sequence number is generated by a reference to the sequence pseudocolumn NEXTVAL, while the current sequence number can be repeatedly referenced using the pseudo-column CURRVAL.
- Caching Sequence Numbers
  Caching sequence numbers can improve access time.
- Making a Sequence Scalable
  A sequence can be made *scalable* by specifying the SCALE clause in the CREATE SEQUENCE or ALTER SEQUENCE statement.

23.2.4.1 Referencing a Sequence

A sequence is referenced in SQL statements with the `NEXTVAL` and `CURRVAL` pseudocolumns; each new sequence number is generated by a reference to the sequence pseudocolumn `NEXTVAL`, while the current sequence number can be repeatedly referenced using the pseudo-column `CURRVAL`.

`NEXTVAL` and `CURRVAL` are not reserved words or keywords and can be used as pseudocolumn names in SQL statements such as `SELECT`, `INSERT`, or `UPDATE`.

- Generating Sequence Numbers with NEXTVAL
  To generate and use a sequence number, reference *seq_name*.`NEXTVAL` in a SQL statement.
- Using Sequence Numbers with CURRVAL
  To use or refer to the current sequence value of your session, reference *seq_name*.`CURRVAL` in a SQL statement.
- Uses and Restrictions of NEXTVAL and CURRVAL
  `CURRVAL` and `NEXTVAL` can be used in specific places, and restrictions apply to their use.

23.2.4.1.1 Generating Sequence Numbers with NEXTVAL

To generate and use a sequence number, reference *seq_name*.`NEXTVAL` in a SQL statement.

For example, assume a customer places an order. The sequence number can be referenced in a values list. For example:

```
INSERT INTO Orders_tab (Orderno, Custno)
    VALUES (Order_seq.NEXTVAL, 1032);
```

Or, the sequence number can be referenced in the `SET` clause of an `UPDATE` statement. For example:

```
UPDATE Orders_tab
    SET Orderno = Order_seq.NEXTVAL
    WHERE Orderno = 10112;
```

The sequence number can also be referenced outermost `SELECT` of a query or subquery. For example:

```
SELECT Order_seq.NEXTVAL FROM dual;
```

As defined, the first reference to `order_seq.NEXTVAL` returns the value 1. Each subsequent statement that references `order_seq.NEXTVAL` generates the next sequence number (2, 3, 4,. . .). The pseudo-column `NEXTVAL` can be used to generate as many new sequence numbers as necessary. However, only a single sequence number can be generated for each row. In other words, if `NEXTVAL` is referenced more than once in a single statement, then the first reference generates the next number, and all subsequent references in the statement return the same number.

Once a sequence number is generated, the sequence number is available only to the session that generated the number. Independent of transactions committing or rolling back, other users referencing `order_seq.NEXTVAL` obtain unique values. If two users are accessing the same sequence concurrently, then the sequence numbers each user receives might have gaps because sequence numbers are also being generated by the other user.

23.2.4.1.2 Using Sequence Numbers with CURRVAL

To use or refer to the current sequence value of your session, reference *seq_name*.CURRVAL in a SQL statement.

CURRVAL can only be used if *seq_name*.NEXTVAL has been referenced in the current user session (in the current or a previous transaction). CURRVAL can be referenced as many times as necessary, including multiple times within the same statement. The next sequence number is not generated until NEXTVAL is referenced. Continuing with the previous example, you would finish placing the customer's order by inserting the line items for the order:

```
INSERT INTO Line_items_tab (Orderno, Partno, Quantity)
    VALUES (Order_seq.CURRVAL, 20321, 3);

INSERT INTO Line_items_tab (Orderno, Partno, Quantity)
    VALUES (Order_seq.CURRVAL, 29374, 1);
```

Assuming the INSERT statement given in the previous section generated a new sequence number of 347, both rows inserted by the statements in this section insert rows with order numbers of 347.

23.2.4.1.3 Uses and Restrictions of NEXTVAL and CURRVAL

CURRVAL and NEXTVAL can be used in specific places, and restrictions apply to their use.

CURRVAL and NEXTVAL can be used in the following places:

- VALUES clause of INSERT statements
- The SELECT list of a SELECT statement
- A view query or materialized view query

  However, the use of CURRVAL and NEXTVAL in a materialized view query makes the materialized view complex. Therefore, it cannot be fast refreshed.
- The SET clause of an UPDATE statement

CURRVAL and NEXTVAL cannot be used in these places:

- A subquery
- A SELECT statement with the DISTINCT operator
- A SELECT statement with a GROUP BY or ORDER BY clause
- A SELECT statement that is combined with another SELECT statement with the UNION, INTERSECT, or MINUS set operator
- The WHERE clause of a SELECT statement
- The condition of a CHECK constraint

23.2.4.2 Caching Sequence Numbers

Caching sequence numbers can improve access time.

- About Caching Sequence Numbers
  Sequence numbers can be kept in the sequence cache in the System Global Area (SGA). Sequence numbers can be accessed more quickly in the sequence cache than they can be read from disk.

- About Automatic Sizing of the Sequence Cache
  Automatic resizing of the sequence cache improves performance significantly for fast insert workloads that use sequences.
- The Number of Entries in the Sequence Cache
  When an application accesses a sequence in the sequence cache, the sequence numbers are read quickly. However, if an application accesses a sequence that is not in the cache, then the sequence must be read from disk to the cache before the sequence numbers are used.
- The Number of Values in Each Sequence Cache Entry
  When a sequence is read into the sequence cache, sequence values are generated and stored in a cache entry. These values can then be accessed quickly.

23.2.4.2.1 About Caching Sequence Numbers

Sequence numbers can be kept in the sequence cache in the System Global Area (SGA). Sequence numbers can be accessed more quickly in the sequence cache than they can be read from disk.

The sequence cache consists of entries. Each entry can hold many sequence numbers for a single sequence.

Follow these guidelines for fast access to all sequence numbers:

- Be sure the sequence cache can hold all the sequences used concurrently by your applications.
- Increase the number of values for each sequence held in the sequence cache.

23.2.4.2.2 About Automatic Sizing of the Sequence Cache

Automatic resizing of the sequence cache improves performance significantly for fast insert workloads that use sequences.

The automatic sequence cache size on each instance is dynamically computed based on the rate of usage of sequence numbers. Each instance caches the maximum of the manually configured sequence cache size and the projected cache size requirement for the next 10 seconds. Based on the sequence usage, the sequence cache size can shrink or grow. The minimum size to which the cache can shrink is the manually configured cache size. To prevent the sequence cache size from growing indefinitely, the cache size and each increment in the cache size is capped.

For cycle sequences, the upper bound for the automatic sequence cache size is the size of one cycle.

23.2.4.2.3 The Number of Entries in the Sequence Cache

When an application accesses a sequence in the sequence cache, the sequence numbers are read quickly. However, if an application accesses a sequence that is not in the cache, then the sequence must be read from disk to the cache before the sequence numbers are used.

If your applications use many sequences concurrently, then your sequence cache might not be large enough to hold all the sequences. In this case, access to sequence numbers might often require disk reads. For fast access to all sequences, be sure your cache has enough entries to hold all the sequences used concurrently by your applications.

Automatic Sizing of Sequence Cache

Starting with Oracle Database 21c, the size of the sequence cache on each instance is dynamically computed. The automatic sequence cache size is based on the rate of usage of sequence numbers. Each instance caches the maximum of the manually configured sequence cache size and the projected cache size requirement for the next 10 seconds. Based on the sequence usage, the sequence cache size can shrink or grow. To prevent the sequence cache size from growing indefinitely, the cache size and each increment in the cache size is capped.

Restrictions on automatic sequence cache size include the following:

- For cycle sequences, the upper bound for the automatic sequence cache size is the size of one cycle.

  .

- Automatic sequence caching is not available for ordered sequences on Oracle Real Application Clusters (Oracle RAC).

> **Note:**
>
> Automatic tuning of sequence cache sizes is available with Oracle Autonomous Database only.

23.2.4.2.4 The Number of Values in Each Sequence Cache Entry

When a sequence is read into the sequence cache, sequence values are generated and stored in a cache entry. These values can then be accessed quickly.

The number of sequence values stored in the cache is determined by the `CACHE` parameter in the `CREATE SEQUENCE` statement. The default value for this parameter is 20.

This `CREATE SEQUENCE` statement creates the `seq2` sequence so that 50 values of the sequence are stored in the `SEQUENCE` cache:

```
CREATE SEQUENCE seq2
    CACHE 50;
```

The first 50 values of `seq2` can then be read from the cache. When the 51st value is accessed, the next 50 values will be read from disk.

Choosing a high value for `CACHE` lets you access more successive sequence numbers with fewer reads from disk to the sequence cache. However, if there is an instance failure, then all sequence values in the cache are lost. Cached sequence numbers also could be skipped after an export and import if transactions continue to access the sequence numbers while the export is running.

If you use the `NOCACHE` option in the `CREATE SEQUENCE` statement, then the values of the sequence are not stored in the sequence cache. In this case, every access to the sequence requires a disk read. Such disk reads slow access to the sequence. This `CREATE SEQUENCE` statement creates the `SEQ3` sequence so that its values are never stored in the cache:

```
CREATE SEQUENCE seq3
    NOCACHE;
```

23.2.4.3 Making a Sequence Scalable

A sequence can be made *scalable* by specifying the `SCALE` clause in the `CREATE SEQUENCE` or `ALTER SEQUENCE` statement.

A scalable sequence is particularly efficient when used to generate unordered primary or unique keys for data ingestion workloads having high level of concurrency. Single Oracle database instances as well as Oracle RAC databases benefit from this feature. Scalable sequences significantly reduce the sequence and index block contention and provide better data load scalability compared to the solution of configuring a very large sequence cache using the `CACHE` clause of `CREATE SEQUENCE` or `ALTER SEQUENCE` statement.

> **Note:**
>
> In addition to using a scalable sequence, you can also partition the data to increase the performance of a data load operation.

The following is the syntax for defining a scalable sequence:

```
CREATE | ALTER SEQUENCE sequence_name
   ...
   SCALE [EXTEND | NOEXTEND] | NOSCALE
   ...
```

When the `SCALE` clause is specified, a 6 digit numeric *scalable sequence offset number* is prefixed to the digits of the sequence:

```
   scalable sequence number = 6 digit scalable sequence offset number ||
normal sequence number
```

where,

- `||` is the concatenation operator.
- 6 digit scalable sequence offset number = 3 digit *instance* offset number `||` 3 digit *session* offset number.

  The 3 digit *instance* offset number is generated as [`(instance id % 100) + 100`]. The 3 digit *session* offset number is generated as [`session id % 1000`].

Additionally, you can also specify `EXTEND` or `NOEXTEND` option for the `SCALE` clause:

- `EXTEND` option

  When the `EXTEND` option is specified for the `SCALE` clause, the scalable sequence values are of the length [`X` digits + `Y` digits], where `X` is the number of digits in the scalable sequence offset number (default is 6 digits), and `Y` is the number of digits specified in the `MAXVALUE` clause.

For example, for an ascending scalable sequence with `MINVALUE` of 1, `MAXVALUE` of 100 (3 digits), and `EXTEND` option specified, the scalable sequence values will be of 9 digits (6 digit scalable sequence offset number + 3 digit `MAXVALUE`) and will be of the form:

```
6 digit scalable sequence offset number || 001
6 digit scalable sequence offset number || 002
6 digit scalable sequence offset number || 003
...
6 digit scalable sequence offset number || 100
```

- `NOEXTEND` option

  When the `NOEXTEND` option is specified for the `SCALE` clause, which is the default option, the number of scalable sequence digits cannot exceed the number of digits specified in the `MAXVALUE` clause.

  For example, for an ascending scalable sequence with `MINVALUE` of 1, `MAXVALUE` of 1000000 (7 digits), and `NOEXTEND` option specified, the scalable sequence values will be of 7 digits, because `MAXVALUE` of 1000000 contains 7 digits, and will be of the form:

  ```
  6 digit scalable sequence offset number || 1
  6 digit scalable sequence offset number || 2
  6 digit scalable sequence offset number || 3
  ...
  6 digit scalable sequence offset number || 9
  ```

  Note that the `NEXTVAL` operation on this scalable sequence after the sequence value of `[6 digit scalable sequence offset number || 9]` will report the following error message, because the next scalable sequence value is `[6 digit scalable sequence offset number || 10]`, which contains 8 digits and is greater than `MAXVALUE` of 1000000 that contains 7 digits:

  ```
  ORA-64603: NEXTVAL cannot be instantiated for SQ. Widen the sequence by 1
  digits or alter sequence with SCALE EXTEND.
  ```

  > **Note:**
  >
  > The `NOEXTEND` option is useful for integration with the existing applications where sequences are used to populate fixed width columns.

To convert an existing scalable sequence to a non-scalable sequence, use the `NOSCALE` clause in the `ALTER SEQUENCE` statement.

> **Note:**
>
> Oracle recommends that you should not specify ordering for a scalable sequence, because scalable sequence numbers are globally unordered.

To know whether a sequence is scalable or whether a scalable sequence is extendable, check the values of the following columns of the DBA_SEQUENCES, USER_SEQUENCES, and ALL_SEQUENCES views.

Table 23-1 Columns Related to Scalable Sequences in the DBA_SEQUENCES, USER_SEQUENCES, and ALL_SEQUENCES Views

Column Name	Description
SCALE_FLAG	Indicates whether the sequence is a scalable sequence: • Y • N
EXTEND_FLAG	Indicates whether the scalable sequence is *extendable*, that is, whether the EXTEND option is applied for the scalable sequence, so that sequence values can extend beyond the value specified for MAXVALUE: • Y • N

23.2.5 Dropping Sequences

If a sequence is no longer required, you can drop the sequence using the DROP SEQUENCE statement.

You can drop any sequence in your schema. To drop a sequence in another schema, you must have the DROP ANY SEQUENCE system privilege. For example, the following statement drops the order_seq sequence:

```
DROP SEQUENCE order_seq;
```

When a sequence is dropped, its definition is removed from the data dictionary. Any synonyms for the sequence remain, but return an error when referenced.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for syntax and additional information about the DROP SEQUENCE statement

23.3 Managing Synonyms

You can perform tasks such as creating synonyms, using synonyms, and dropping synonyms.

- About Synonyms
  A synonym is an alias for a schema object.
- Creating Synonyms
  Create a synonym using the CREATE SYNONYM statement.
- Using Synonyms in DML Statements
  A synonym can be referenced in a DML statement the same way that the underlying object of the synonym can be referenced.

- Dropping Synonyms
  Drop a synonym that is no longer required using `DROP SYNONYM` statement. To drop a private synonym, omit the `PUBLIC` keyword. To drop a public synonym, include the `PUBLIC` keyword.

23.3.1 About Synonyms

A synonym is an alias for a schema object.

Synonyms can provide a level of security by masking the name and owner of an object and by providing location transparency for remote objects of a distributed database. Also, they are convenient to use and reduce the complexity of SQL statements for database users.

Synonyms allow underlying objects to be renamed or moved, where only the synonym must be redefined and applications based on the synonym continue to function without modification.

You can create both public and private synonyms. A **public** synonym is owned by the special user group named `PUBLIC` and is accessible to every user in a database. A **private** synonym is contained in the schema of a specific user and available only to the user and to grantees for the underlying object.

Synonyms themselves are not securable. When you grant object privileges on a synonym, you are really granting privileges on the underlying object, and the synonym is acting only as an alias for the object in the `GRANT` statement.

> **See Also:**
>
> *Oracle Database Concepts* for a more complete description of synonyms

23.3.2 Creating Synonyms

Create a synonym using the `CREATE SYNONYM` statement.

To create a private synonym in your own schema, you must have the `CREATE SYNONYM` privilege. To create a private synonym in another user's schema, you must have the `CREATE ANY SYNONYM` privilege. To create a public synonym, you must have the `CREATE PUBLIC SYNONYM` system privilege.

When you create a synonym, the underlying schema object need not exist, nor do you need privileges to access the object for the `CREATE SYNONYM` statement to succeed. The following statement creates a public synonym named `public_emp` on the `emp` table contained in the schema of `jward`:

```
CREATE PUBLIC SYNONYM public_emp FOR jward.emp
```

When you create a synonym for a remote procedure or function, you must qualify the remote object with its schema name. Alternatively, you can create a local public synonym on the database where the remote object resides, in which case the database link must be included in all subsequent calls to the procedure or function.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for syntax and additional information about the `CREATE SYNONYM` statement

23.3.3 Using Synonyms in DML Statements

A synonym can be referenced in a DML statement the same way that the underlying object of the synonym can be referenced.

You can successfully use any private synonym contained in your schema or any public synonym, assuming that you have the necessary privileges to access the underlying object, either explicitly, from an enabled role, or from `PUBLIC`. You can also reference any private synonym contained in another schema if you have been granted the necessary object privileges for the underlying object.

You can reference another user's synonym using only the object privileges that you have been granted. For example, if you have only the `SELECT` privilege on the `jward.emp` table, and the synonym `jward.employee` is created for `jward.emp`, you can query the `jward.employee` synonym, but you cannot insert rows using the `jward.employee` synonym.

For example, if a synonym named `emp`loyee refers to a table or view, then the following statement is valid:

```
INSERT INTO employee (empno, ename, job)
    VALUES (emp_sequence.NEXTVAL, 'SMITH', 'CLERK');
```

If the synonym named `fire_emp` refers to a standalone procedure or package procedure, then you could execute it with the command

```
EXECUTE Fire_emp(7344);
```

23.3.4 Dropping Synonyms

Drop a synonym that is no longer required using `DROP SYNONYM` statement. To drop a private synonym, omit the `PUBLIC` keyword. To drop a public synonym, include the `PUBLIC` keyword.

You can drop any private synonym in your own schema. To drop a private synonym in another user's schema, you must have the `DROP ANY SYNONYM` system privilege. To drop a public synonym, you must have the `DROP PUBLIC SYNONYM` system privilege.

For example, the following statement drops the private synonym named `emp`:

```
DROP SYNONYM emp;
```

The following statement drops the public synonym named `public_emp`:

```
DROP PUBLIC SYNONYM public_emp;
```

When you drop a synonym, its definition is removed from the data dictionary. All objects that reference a dropped synonym remain. However, they become invalid (not usable). For more information about how dropping synonyms can affect other schema objects, see "Managing Object Dependencies".

> **See Also:**
>
> *Oracle Database SQL Language Reference* for syntax and additional information about the `DROP SYNONYM` statement

23.4 Views, Synonyms, and Sequences Data Dictionary Views

You can query data dictionary views for information about views, synonyms, and sequences.

The following views display information about views, synonyms, and sequences:

View	Description

`DBA_VIEWS`
`ALL_VIEWS`
`USER_VIEWS`
	`DBA` view describes all views in the database. `ALL` view is restricted to views accessible to the current user. `USER` view is restricted to views owned by the current user.

`DBA_SYNONYMS`
`ALL_SYNONYMS`
`USER_SYNONYMS`
	These views describe synonyms.

`DBA_SEQUENCES`
`ALL_SEQUENCES`
`USER_SEQUENCES`
	These views describe sequences.

`DBA_UPDATABLE_COLUMNS`
`ALL_UPDATABLE_COLUMNS`
`USER_UPDATABLE_COLUMNS`
	These views describe all columns in join views that are updatable.