17

Managing Schema Objects

You can create and manage several types of schema objects with Oracle Database.

- About Common and Local Objects
  A **common object** is defined in either the CDB root or an application root, and can be referenced using metadata links or object links. A local object is every object that is not a common object.
- About the Container for Schema Objects
  Schema objects are created in the current container.
- Creating Multiple Tables and Views in a Single Operation
  You can create several tables and views and grant privileges in one operation using the `CREATE SCHEMA` statement. If an individual table, view or grant fails, the entire statement is rolled back. None of the objects are created, nor are the privileges granted.
- Analyzing Tables, Indexes, and Clusters
  You can collecting statistics on schema objects, analyze the statistics, and validate the schema objects.
- Truncating Tables and Clusters
  You can delete all rows of a table or all rows in a group of clustered tables so that the table (or cluster) still exists, but is completely empty. For example, consider a table that contains monthly data, and at the end of each month, you must empty it (delete all rows) after archiving its data.
- Enabling and Disabling Triggers
  Database triggers are procedures that are stored in the database and activated ("fired") when specific conditions occur, such as adding a row to a table.
- Managing Integrity Constraints
  Integrity constraints are rules that restrict the values for one or more columns in a table. Constraint clauses can appear in either `CREATE TABLE` or `ALTER TABLE` statements, and identify the column or columns affected by the constraint and identify the conditions of the constraint.
- Renaming Schema Objects
  There are several ways to rename an object.
- Managing Object Dependencies
  Oracle Database provides an automatic mechanism to ensure that a dependent object is always up to date with respect to its referenced objects. You can also manually recompile invalid object.
- Managing Object Name Resolution
  Object names referenced in SQL statements can consist of several pieces, separated by periods. Oracle Database performs specific actions to resolve an object name.
- Switching to a Different Schema
  Use an `ALTER SESSION` SQL statement to switch to a different schema.
- Managing Editions
  Application developers who are upgrading their applications using edition-based redefinition may ask you to perform edition-related tasks that require DBA privileges.

- Displaying Information About Schema Objects
  Oracle Database provides a PL/SQL package that enables you to determine the DDL that created an object and data dictionary views that you can use to display information about schema objects.

17.1 About Common and Local Objects

A **common object** is defined in either the CDB root or an application root, and can be referenced using metadata links or object links. A local object is every object that is not a common object.

Database-supplied common objects are defined in `CDB$ROOT` and cannot be changed. Oracle Database does not support creation of common objects in `CDB$ROOT`.

You can create most schema objects—such as tables, views, PL/SQL and Java program units, sequences, and so on—as common objects in an application root. If the object exists in an application root, then it is called an **application common object**.

A local user can own a common object. Also, a common user can own a local object, but only when the object is not data-linked or metadata-linked, and is also neither a metadata link nor a data link.

> **See Also:**
>
> *Oracle Database Security Guide* to learn more about privilege management for common objects

17.2 About the Container for Schema Objects

Schema objects are created in the current container.

Before you create schema objects, ensure that you are in the container that store these schema objects.

To create a schema object in a pluggable database (PDB), connect to the PDB as a common user or local user with the required privileges. Then, run the required SQL*Plus command.

17.3 Creating Multiple Tables and Views in a Single Operation

You can create several tables and views and grant privileges in one operation using the `CREATE SCHEMA` statement. If an individual table, view or grant fails, the entire statement is rolled back. None of the objects are created, nor are the privileges granted.

Specifically, the `CREATE SCHEMA` statement can include *only* `CREATE TABLE`, `CREATE VIEW`, and `GRANT` statements. You must have the privileges necessary to issue the included statements. You are not actually creating a schema, that is done when the user is created with a `CREATE USER` statement. Rather, you are populating the schema.

The following statement creates two tables and a view that joins data from the two tables:

```
CREATE SCHEMA AUTHORIZATION scott
    CREATE TABLE dept (
```

```
    deptno NUMBER(3,0) PRIMARY KEY,
    dname VARCHAR2(15),
    loc VARCHAR2(25))
CREATE TABLE emp (
    empno NUMBER(5,0) PRIMARY KEY,
    ename VARCHAR2(15) NOT NULL,
    job VARCHAR2(10),
    mgr NUMBER(5,0),
    hiredate DATE DEFAULT (sysdate),
    sal NUMBER(7,2),
    comm NUMBER(7,2),
    deptno NUMBER(3,0) NOT NULL
    CONSTRAINT dept_fkey REFERENCES dept)
CREATE VIEW sales_staff AS
    SELECT empno, ename, sal, comm
    FROM emp
    WHERE deptno = 30
    WITH CHECK OPTION CONSTRAINT sales_staff_cnst
    GRANT SELECT ON sales_staff TO human_resources;
```

The `CREATE SCHEMA` statement does not support Oracle Database extensions to the ANSI `CREATE TABLE` and `CREATE VIEW` statements, including the `STORAGE` clause.

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

17.4 Analyzing Tables, Indexes, and Clusters

You can collecting statistics on schema objects, analyze the statistics, and validate the schema objects.

- About Analyzing Tables, Indexes, and Clusters
  You can collect information about schema objects and analyze that information.
- Using DBMS_STATS to Collect Table and Index Statistics
  You can use the `DBMS_STATS` package or the `ANALYZE` statement to gather statistics about the physical storage characteristics of a table, index, or cluster. These statistics are stored in the data dictionary and can be used by the optimizer to choose the most efficient execution plan for SQL statements accessing analyzed objects.
- Validating Tables, Indexes, Clusters, and Materialized Views
  To verify the integrity of the structure of a table, index, cluster, or materialized view, use the `ANALYZE` statement with the `VALIDATE STRUCTURE` option.
- Cross Validation of a Table and an Index with a Query
  In some cases, an `ANALYZE` statement takes an inordinate amount of time to complete. In these cases, you can use a SQL query to validate an index.
- Listing Chained Rows of Tables and Clusters
  You can look at the chained and migrated rows of a table or cluster using the `ANALYZE` statement with the `LIST CHAINED ROWS` clause. The results of this statement are stored in a specified table created explicitly to accept the information returned by the `LIST CHAINED ROWS` clause. These results are useful in determining whether you have enough room for updates to rows.

17.4.1 About Analyzing Tables, Indexes, and Clusters

You can collect information about schema objects and analyze that information.

You analyze a schema object (table, index, or cluster) to:

- Collect and manage statistics for it
- Verify the validity of its storage format
- Identify migrated and chained rows of a table or cluster

> **Note:**
>
> Do not use the `COMPUTE` and `ESTIMATE` clauses of `ANALYZE` to collect optimizer statistics. These clauses have been deprecated. Instead, use the `DBMS_STATS` package, which lets you collect statistics in parallel, collect global statistics for partitioned objects, and fine tune your statistics collection in other ways. The cost-based optimizer, which depends upon statistics, will eventually use only statistics that have been collected by `DBMS_STATS`. See *Oracle Database PL/SQL Packages and Types Reference* for more information on the `DBMS_STATS` package.
>
> You must use the `ANALYZE` statement (rather than `DBMS_STATS`) for statistics collection not related to the cost-based optimizer, such as:
>
> - To use the `VALIDATE` or `LIST CHAINED ROWS` clauses
> - To collect information on freelist blocks

17.4.2 Using DBMS_STATS to Collect Table and Index Statistics

You can use the `DBMS_STATS` package or the `ANALYZE` statement to gather statistics about the physical storage characteristics of a table, index, or cluster. These statistics are stored in the data dictionary and can be used by the optimizer to choose the most efficient execution plan for SQL statements accessing analyzed objects.

Oracle recommends using the more versatile `DBMS_STATS` package for gathering optimizer statistics, but you must use the `ANALYZE` statement to collect statistics unrelated to the optimizer, such as empty blocks, average space, and so forth.

The `DBMS_STATS` package allows both the gathering of statistics, including utilizing parallel execution, and the external manipulation of statistics. Statistics can be stored in tables outside of the data dictionary, where they can be manipulated without affecting the optimizer. Statistics can be copied between databases or backup copies can be made.

The following `DBMS_STATS` procedures enable the gathering of optimizer statistics:

- `GATHER_INDEX_STATS`
- `GATHER_TABLE_STATS`
- `GATHER_SCHEMA_STATS`
- `GATHER_DATABASE_STATS`

> **See Also:**
>
> - *Oracle Database SQL Tuning Guide* for information about using `DBMS_STATS` to gather statistics for the optimizer
> - *Oracle Database PL/SQL Packages and Types Reference* for a description of the `DBMS_STATS` package

17.4.3 Validating Tables, Indexes, Clusters, and Materialized Views

To verify the integrity of the structure of a table, index, cluster, or materialized view, use the `ANALYZE` statement with the `VALIDATE STRUCTURE` option.

If the structure is valid, then no error is returned. However, if the structure is corrupt, then you receive an error message.

For example, in rare cases such as hardware or other system failures, an index can become corrupted and not perform correctly. When validating the index, you can confirm that every entry in the index points to the correct row of the associated table. If the index is corrupt, then you can drop and re-create it.

If a table, index, or cluster is corrupt, then drop it and re-create it. If a materialized view is corrupt, then perform a complete refresh and ensure that you have remedied the problem. If the problem is not corrected, then drop and re-create the materialized view.

The following statement analyzes the `emp` table:

```
ANALYZE TABLE emp VALIDATE STRUCTURE;
```

You can validate an object and all dependent objects (for example, indexes) by including the `CASCADE` option. The following statement validates the `emp` table and all associated indexes:

```
ANALYZE TABLE emp VALIDATE STRUCTURE CASCADE;
```

By default the `CASCADE` option performs a complete validation. Because this operation can be resource intensive, you can perform a faster version of the validation by using the `FAST` clause. This version checks for the existence of corruptions using an optimized check algorithm, but does not report details about the corruption. If the `FAST` check finds a corruption, then you can then use the `CASCADE` option without the `FAST` clause to locate it. The following statement performs a fast validation on the `emp` table and all associated indexes:

```
ANALYZE TABLE emp VALIDATE STRUCTURE CASCADE FAST;
```

If fast validation takes an inordinate amount of time, then you have the option of validating individual indexes with a SQL query. See "Cross Validation of a Table and an Index with a Query".

You can specify that you want to perform structure validation online while DML is occurring against the object being validated. Validation is less comprehensive with ongoing DML affecting the object, but this is offset by the flexibility of being able to perform `ANALYZE` online. The following statement validates the `emp` table and all associated indexes online:

```
ANALYZE TABLE emp VALIDATE STRUCTURE CASCADE ONLINE;
```

> **See Also:**
>
> *Oracle Database SQL Language Reference* for more information on the `ANALYZE` statement

17.4.4 Cross Validation of a Table and an Index with a Query

In some cases, an `ANALYZE` statement takes an inordinate amount of time to complete. In these cases, you can use a SQL query to validate an index.

If the query determines that there is an inconsistency between a table and an index, then you can use an `ANALYZE` statement for a thorough analysis of the index. Since typically most objects in a database are not corrupt, you can use this quick query to eliminate a number of tables as candidates for corruption and only use the ANALYZE statement on tables that might be corrupt.

To validate an index, run the following query:

```
SELECT /*+ FULL(ALIAS) PARALLEL(ALIAS, DOP) */ SUM(ORA_HASH(ROWID))
   FROM table_name ALIAS
   WHERE ALIAS.index_column IS NOT NULL
      MINUS SELECT /*+ INDEX_FFS(ALIAS index_name)
      PARALLEL_INDEX(ALIAS, index_name, DOP) */ SUM(ORA_HASH(ROWID))
   FROM table_name ALIAS WHERE ALIAS.index_column IS NOT NULL;
```

When you run the query, make the following substitutions:

- Enter the table name for the *table_name* placeholder.
- Enter the index column for the *index_column* placeholder.
- Enter the index name for the *index_name* placeholder.

If the query returns any rows, then there is a possible inconsistency, and you can use an `ANALYZE` statement for further diagnosis.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for more information about the `ANALYZE` statement

17.4.5 Listing Chained Rows of Tables and Clusters

You can look at the chained and migrated rows of a table or cluster using the `ANALYZE` statement with the `LIST CHAINED ROWS` clause. The results of this statement are stored in a specified table created explicitly to accept the information returned by the `LIST CHAINED ROWS` clause. These results are useful in determining whether you have enough room for updates to rows.

- Creating a CHAINED_ROWS Table
  To create the table to accept data returned by an `ANALYZE...LIST CHAINED ROWS` statement, execute the `UTLCHAIN.SQL` or `UTLCHN1.SQL` script.

- Eliminating Migrated or Chained Rows in a Table
  You can use the information in the CHAINED_ROWS table to reduce or eliminate migrated and chained rows in an existing table.

17.4.5.1 Creating a CHAINED_ROWS Table

To create the table to accept data returned by an ANALYZE...LIST CHAINED ROWS statement, execute the UTLCHAIN.SQL or UTLCHN1.SQL script.

These scripts are provided by the database. They create a table named CHAINED_ROWS in the schema of the user submitting the script.

> **Note:**
>
> Your choice of script to execute for creating the CHAINED_ROWS table depends on the compatibility level of your database and the type of table you are analyzing. See the *Oracle Database SQL Language Reference* for more information.

After a CHAINED_ROWS table is created, you specify it in the INTO clause of the ANALYZE statement. For example, the following statement inserts rows containing information about the chained rows in the emp_dept cluster into the CHAINED_ROWS table:

```
ANALYZE CLUSTER emp_dept LIST CHAINED ROWS INTO CHAINED_ROWS;
```

> **See Also:**
>
> - *Oracle Database Reference* for a description of the CHAINED_ROWS table
> - "Using the Segment Advisor" for information on how the Segment Advisor reports tables with excess row chaining.

17.4.5.2 Eliminating Migrated or Chained Rows in a Table

You can use the information in the CHAINED_ROWS table to reduce or eliminate migrated and chained rows in an existing table.

Use the following procedure:

1. Use the ANALYZE statement to collect information about migrated and chained rows.

   ```
   ANALYZE TABLE order_hist LIST CHAINED ROWS;
   ```

2. Query the output table:

   ```
   SELECT *
   FROM CHAINED_ROWS
   WHERE TABLE_NAME = 'ORDER_HIST';

   OWNER_NAME  TABLE_NAME  CLUST... HEAD_ROWID          TIMESTAMP
   ----------  ----------  -----... ------------------  ---------
   SCOTT       ORDER_HIST       ... AAAAluAAHAAAAA1AAA  04-MAR-96
   ```

```
SCOTT       ORDER_HIST       ... AAAAluAAHAAAAA1AAB  04-MAR-96
SCOTT       ORDER_HIST       ... AAAAluAAHAAAAA1AAC  04-MAR-96
```

The output lists all rows that are either migrated or chained.

3. If the output table shows that you have many migrated or chained rows, then you can eliminate migrated rows by continuing through the following steps:
4. Create an intermediate table with the same columns as the existing table to hold the migrated and chained rows:

```
CREATE TABLE int_order_hist
   AS SELECT *
      FROM order_hist
      WHERE ROWID IN
         (SELECT HEAD_ROWID
            FROM CHAINED_ROWS
            WHERE TABLE_NAME = 'ORDER_HIST');
```

5. Delete the migrated and chained rows from the existing table:

```
DELETE FROM order_hist
   WHERE ROWID IN
      (SELECT HEAD_ROWID
         FROM CHAINED_ROWS
         WHERE TABLE_NAME = 'ORDER_HIST');
```

6. Insert the rows of the intermediate table into the existing table:

```
INSERT INTO order_hist
   SELECT *
   FROM int_order_hist;
```

7. Drop the intermediate table:

```
DROP TABLE int_order_history;
```

8. Delete the information collected in step 1 from the output table:

```
DELETE FROM CHAINED_ROWS
   WHERE TABLE_NAME = 'ORDER_HIST';
```

9. Use the `ANALYZE` statement again, and query the output table.

Any rows that appear in the output table are chained. You can eliminate chained rows only by increasing your data block size. It might not be possible to avoid chaining in all situations. Chaining is often unavoidable with tables that have a `LONG` column or large `CHAR` or `VARCHAR2` columns.

17.5 Truncating Tables and Clusters

You can delete all rows of a table or all rows in a group of clustered tables so that the table (or cluster) still exists, but is completely empty. For example, consider a table that contains monthly data, and at the end of each month, you must empty it (delete all rows) after archiving its data.

- Using DELETE to Truncate a Table
  You can delete the rows of a table using the `DELETE` SQL statement.
- Using DROP and CREATE to Truncate a Table
  You can drop a table and then re-create the table to truncate it.
- Using TRUNCATE
  You can delete all rows of the table using the `TRUNCATE` statement.

17.5.1 Using DELETE to Truncate a Table

You can delete the rows of a table using the `DELETE` SQL statement.

For example, the following statement deletes all rows from the `emp` table:

```
DELETE FROM emp;
```

If there are many rows present in a table or cluster when using the `DELETE` statement, significant system resources are consumed as the rows are deleted. For example, CPU time, redo log space, and undo segment space from the table and any associated indexes require resources. Also, as each row is deleted, triggers can be fired. The space previously allocated to the resulting empty table or cluster remains associated with that object. With `DELETE` you can choose which rows to delete, whereas `TRUNCATE` and `DROP` affect the entire object.

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

17.5.2 Using DROP and CREATE to Truncate a Table

You can drop a table and then re-create the table to truncate it.

For example, the following statements drop and then re-create the `emp` table:

```
DROP TABLE emp;
CREATE TABLE emp ( ... );
```

When dropping and re-creating a table or cluster, all associated indexes, integrity constraints, and triggers are also dropped, and all objects that depend on the dropped table or clustered table are invalidated. Also, all grants for the dropped table or clustered table are dropped.

17.5.3 Using TRUNCATE

You can delete all rows of the table using the `TRUNCATE` statement.

For example, the following statement truncates the `emp` table:

```
TRUNCATE TABLE emp;
```

Using the `TRUNCATE` statement provides a fast, efficient method for deleting all rows from a table or cluster. A `TRUNCATE` statement does not generate any undo information and it commits immediately. It is a DDL statement and cannot be rolled back. A `TRUNCATE` statement does not affect any structures associated with the table being truncated (constraints and triggers) or authorizations. A `TRUNCATE` statement also specifies whether space currently allocated for the table is returned to the containing tablespace after truncation.

You can truncate any table or cluster in your own schema. Any user who has the `DROP ANY TABLE` system privilege can truncate a table or cluster in any schema.

Before truncating a table or clustered table containing a parent key, all referencing foreign keys in different tables must be disabled. A self-referential constraint does not have to be disabled.

As a TRUNCATE statement deletes rows from a table, triggers associated with the table are not fired. Also, a TRUNCATE statement does not generate any audit information corresponding to DELETE statements if auditing is enabled. Instead, a single audit record is generated for the TRUNCATE statement being issued.

A hash cluster cannot be truncated, nor can tables within a hash or index cluster be individually truncated. Truncation of an index cluster deletes all rows from all tables in the cluster. If all the rows must be deleted from an individual clustered table, use the DELETE statement or drop and re-create the table.

The TRUNCATE statement has several options that control whether space currently allocated for a table or cluster is returned to the containing tablespace after truncation.

These options also apply to any associated indexes. When a table or cluster is truncated, all associated indexes are also truncated. The storage parameters for a truncated table, cluster, or associated indexes are not changed as a result of the truncation.

These TRUNCATE options are:

- DROP STORAGE, the default option, reduces the number of extents allocated to the resulting table to the original setting for MINEXTENTS. Freed extents are then returned to the system and can be used by other objects.
- DROP ALL STORAGE drops the segment. In addition to the TRUNCATE TABLE statement, DROP ALL STORAGE also applies to the ALTER TABLE TRUNCATE (SUB)PARTITION statement. This option also drops any dependent object segments associated with the partition being truncated.

  DROP ALL STORAGE is not supported for clusters.

  ```
  TRUNCATE TABLE emp DROP ALL STORAGE;
  ```

- REUSE STORAGE specifies that all space currently allocated for the table or cluster remains allocated to it. For example, the following statement truncates the emp_dept cluster, leaving all extents previously allocated for the cluster available for subsequent inserts and deletes:

  ```
  TRUNCATE CLUSTER emp_dept REUSE STORAGE;
  ```

See Also:

- *Oracle Database SQL Language Reference* for syntax and other information about the TRUNCATE TABLE statement
- *Oracle Database SQL Language Reference* for syntax and other information about the TRUNCATE CLUSTER statement
- *Oracle Database Security Guide* for information about auditing

17.6 Enabling and Disabling Triggers

Database triggers are procedures that are stored in the database and activated ("fired") when specific conditions occur, such as adding a row to a table.

You can use triggers to supplement the standard capabilities of the database to provide a highly customized database management system. For example, you can create a trigger to

restrict DML operations against a table, allowing only statements issued during regular business hours.

- About Enabling and Disabling Triggers
  An enabled trigger executes its trigger body if a triggering statement is issued and the trigger restriction, if any, evaluates to true. By default, triggers are enabled when first created. A disabled trigger does not execute its trigger body, even if a triggering statement is issued and the trigger restriction (if any) evaluates to true.
- Enabling Triggers
  You enable a disabled trigger using the `ALTER TRIGGER` statement with the `ENABLE` option.
- Disabling Triggers
  You disable a trigger using the `ALTER TRIGGER` statement with the `DISABLE` option.

17.6.1 About Enabling and Disabling Triggers

An enabled trigger executes its trigger body if a triggering statement is issued and the trigger restriction, if any, evaluates to true. By default, triggers are enabled when first created. A disabled trigger does not execute its trigger body, even if a triggering statement is issued and the trigger restriction (if any) evaluates to true.

Database triggers can be associated with a table, schema, or database. They are implicitly fired when:

- DML statements are executed (`INSERT`, `UPDATE`, `DELETE`) against an associated table
- Certain DDL statements are executed (for example: `ALTER`, `CREATE`, `DROP`) on objects within a database or schema
- A specified database event occurs (for example: `STARTUP`, `SHUTDOWN`, `SERVERERROR`)

This is not a complete list. See the *Oracle Database SQL Language Reference* for a full list of statements and database events that cause triggers to fire.

Create triggers with the `CREATE TRIGGER` statement. They can be defined as firing `BEFORE` or `AFTER` the triggering event, or `INSTEAD OF` it. The following statement creates a trigger `scott.emp_permit_changes` on table `scott.emp`. The trigger fires before any of the specified statements are executed.

```
CREATE TRIGGER scott.emp_permit_changes
     BEFORE
     DELETE OR INSERT OR UPDATE
     ON scott.emp
     .
     .
     .
pl/sql block
     .
     .
     .
```

You can later remove a trigger from the database by issuing the `DROP TRIGGER` statement.

To enable or disable triggers using the `ALTER TABLE` statement, you must own the table, have the `ALTER` object privilege for the table, or have the `ALTER ANY TABLE` system privilege. To enable or disable an individual trigger using the `ALTER TRIGGER` statement, you must own the trigger or have the `ALTER ANY TRIGGER` system privilege.

> **See Also:**
>
> - *Oracle Database Concepts* for a more detailed description of triggers
> - *Oracle Database SQL Language Reference* for syntax of the `CREATE TRIGGER` statement
> - *Oracle Database PL/SQL Language Reference* for information about creating and using triggers

17.6.2 Enabling Triggers

You enable a disabled trigger using the `ALTER TRIGGER` statement with the `ENABLE` option.

To enable the disabled trigger named `reorder` on the `inventory` table, enter the following statement:

```
ALTER TRIGGER reorder ENABLE;
```

To enable all triggers defined for a specific table, use the `ALTER TABLE` statement with the `ENABLE ALL TRIGGERS` option. To enable all triggers defined for the `INVENTORY` table, enter the following statement:

```
ALTER TABLE inventory
    ENABLE ALL TRIGGERS;
```

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

17.6.3 Disabling Triggers

You disable a trigger using the `ALTER TRIGGER` statement with the `DISABLE` option.

Consider temporarily disabling a trigger if one of the following conditions is true:

- An object that the trigger references is not available.
- You must perform a large data load and want it to proceed quickly without firing triggers.
- You are loading data into the table to which the trigger applies.

To disable the trigger `reorder` on the `inventory` table, enter the following statement:

```
ALTER TRIGGER reorder DISABLE;
```

You can disable all triggers associated with a table at the same time using the `ALTER TABLE` statement with the `DISABLE ALL TRIGGERS` option. For example, to disable all triggers defined for the `inventory` table, enter the following statement:

```
ALTER TABLE inventory
    DISABLE ALL TRIGGERS;
```

17.7 Managing Integrity Constraints

Integrity constraints are rules that restrict the values for one or more columns in a table. Constraint clauses can appear in either `CREATE TABLE` or `ALTER TABLE` statements, and identify the column or columns affected by the constraint and identify the conditions of the constraint.

- Integrity Constraint States
  Integrity constraints enforce business rules and prevent the entry of invalid information into tables.
- Setting Integrity Constraints Upon Definition
  When an integrity constraint is defined in a `CREATE TABLE` or `ALTER TABLE` statement, it can be enabled, disabled, or validated or not validated as determined by your specification of the `ENABLE`/`DISABLE` clause. If the `ENABLE`/`DISABLE` clause is not specified in a constraint definition, the database automatically enables and validates the constraint.
- Modifying, Renaming, or Dropping Existing Integrity Constraints
  You can use the `ALTER TABLE` statement to enable, disable, modify, or drop a constraint. When the database is using a `UNIQUE` or `PRIMARY KEY` index to enforce a constraint, and constraints associated with that index are dropped or disabled, the index is dropped, unless you specify otherwise.
- Deferring Constraint Checks
  When the database checks a constraint, it signals an error if the constraint is not satisfied. You can defer checking the validity of constraints until the end of a transaction. When you issue the `SET CONSTRAINTS` statement, the `SET CONSTRAINTS` mode lasts for the duration of the transaction, or until another `SET CONSTRAINTS` statement resets the mode.
- Reporting Constraint Exceptions
  If exceptions exist when a constraint is validated, then an error is returned and the integrity constraint remains novalidated. When a statement is not successfully executed because integrity constraint exceptions exist, the statement is rolled back. If exceptions exist, then you cannot validate the constraint until all exceptions to the constraint are either updated or deleted.
- Viewing Constraint Information
  Oracle Database provides a set of views that enable you to see constraint definitions on tables and to identify columns that are specified in constraints.

See Also:

- *Oracle Database Concepts* for a more thorough discussion of integrity constraints
- *Oracle Database Development Guide* for detailed information and examples of using integrity constraints in applications

17.7.1 Integrity Constraint States

Integrity constraints enforce business rules and prevent the entry of invalid information into tables.

17.7.1.1 About Integrity Constraint States

You can specify that a constraint is enabled (ENABLE) or disabled (DISABLE). If a constraint is enabled, data is checked as it is entered or updated in the database, and data that does not conform to the constraint is prevented from being entered. If a constraint is disabled, then data that does not conform can be allowed to enter the database.

Additionally, you can specify that existing data in the table must conform to the constraint (VALIDATE). Conversely, if you specify NOVALIDATE, you are not ensured that existing data conforms.

An integrity constraint defined on a table can be in one of the following states:

- ENABLE, VALIDATE
- ENABLE, NOVALIDATE
- DISABLE, VALIDATE
- DISABLE, NOVALIDATE

For details about the meaning of these states and an understanding of their consequences, see the *Oracle Database SQL Language Reference.* Some of these consequences are discussed here.

17.7.1.2 About Disabling Constraints

To enforce the rules defined by integrity constraints, the constraints should always be enabled, but you can consider disabling them in certain situations.

However, consider temporarily disabling the integrity constraints of a table for the following performance reasons:

- When loading large amounts of data into a table
- When performing batch operations that make massive changes to a table (for example, changing every employee's number by adding 1000 to the existing number)
- When importing or exporting one table at a time

In all three cases, temporarily disabling integrity constraints can improve the performance of the operation, especially in data warehouse configurations.

It is possible to enter data that violates a constraint while that constraint is disabled. Thus, you should always enable the constraint after completing any of the operations listed in the preceding bullet list.

17.7.1.3 About Enabling Constraints

While a constraint is enabled, no row violating the constraint can be inserted into the table.

However, while the constraint is disabled such a row can be inserted. This row is known as an exception to the constraint. If the constraint is in the enable novalidated state, violations resulting from data entered while the constraint was disabled remain. The rows that violate the constraint must be either updated or deleted in order for the constraint to be put in the validated state.

You can identify exceptions to a specific integrity constraint while attempting to enable the constraint. See "Reporting Constraint Exceptions". All rows violating constraints are noted in an `EXCEPTIONS` table, which you can examine.

17.7.1.4 About the Enable Novalidate Constraint State

When a constraint is in the enable novalidate state, all subsequent statements are checked for conformity to the constraint. However, any existing data in the table is not checked.

A table with enable novalidated constraints can contain invalid data, but it is not possible to add new invalid data to it. Enabling constraints in the novalidated state is most useful in data warehouse configurations that are uploading valid OLTP data.

Enabling a constraint does not require validation. Enabling a constraint novalidate is much faster than enabling and validating a constraint. Also, validating a constraint that is already enabled does not require any DML locks during validation (unlike validating a previously disabled constraint). Enforcement guarantees that no violations are introduced during the validation. Hence, enabling without validating enables you to reduce the downtime typically associated with enabling a constraint.

17.7.1.5 Efficient Use of Integrity Constraints: A Procedure

It is important to use integrity constraint states in a particular order.

Using integrity constraint states in the following order can ensure the best benefits:

1. Set state of constraint to DISABLE.
2. Perform the operation (load, export, import).
3. Set state of constraint to ENABLE NOVALIDATE.
4. Set state of constraint to ENABLE.

For example:

```
SQL> CREATE TABLE eg(n NUMBER NOT NULL CONSTRAINT n_gt_0 CHECK (n > 0));

Table created.

SQL> SELECT status AS enabled, validated
     FROM   user_constraints
     WHERE  table_name = 'EG' and constraint_name = 'N_GT_0';

ENABLED  VALIDATED
```

```
-------- -------------
ENABLED  VALIDATED

SQL> ALTER TABLE eg MODIFY CONSTRAINT n_gt_0 DISABLE;

Table altered.

SQL> SELECT status AS enabled, validated
     FROM   user_constraints
     WHERE  table_name = 'EG' and constraint_name = 'N_GT_0';

ENABLED  VALIDATED
-------- -------------
DISABLED NOT VALIDATED

SQL> ALTER TABLE eg MODIFY CONSTRAINT n_gt_0 enable NOVALIDATE;

Table altered.

SQL> SELECT status AS enabled, validated
     FROM   user_constraints
     WHERE  table_name = 'EG' and constraint_name = 'N_GT_0';

ENABLED  VALIDATED
-------- -------------
ENABLED  NOT VALIDATED

SQL> ALTER TABLE eg MODIFY CONSTRAINT n_gt_0 ENABLE;

Table altered.

SQL> SELECT status AS enabled, validated
     FROM   user_constraints
     WHERE  table_name = 'EG' and constraint_name = 'N_GT_0';

ENABLED  VALIDATED
-------- -------------
ENABLED  VALIDATED
```

Some benefits of using constraints in this order are:

- No locks are held.
- All constraints can go to enable state concurrently.
- Constraint enabling is done in parallel.
- Concurrent activity on table is permitted.

The PRIMARY KEY and FOREIGN KEY constraints may not permit concurrent activity due to waits for library cache locks.

17.7.2 Setting Integrity Constraints Upon Definition

When an integrity constraint is defined in a `CREATE TABLE` or `ALTER TABLE` statement, it can be enabled, disabled, or validated or not validated as determined by your specification of the

ENABLE/DISABLE clause. If the ENABLE/DISABLE clause is not specified in a constraint definition, the database automatically enables and validates the constraint.

- Disabling Constraints Upon Definition
  You can disable an integrity constraint when you define it.
- Enabling Constraints Upon Definition
  You can enable an integrity constraint when you define it.

17.7.2.1 Disabling Constraints Upon Definition

You can disable an integrity constraint when you define it.

The following CREATE TABLE and ALTER TABLE statements both define and disable integrity constraints:

```
CREATE TABLE emp (
    empno NUMBER(5) PRIMARY KEY DISABLE,   . . . ;

ALTER TABLE emp
   ADD PRIMARY KEY (empno) DISABLE;
```

An ALTER TABLE statement that defines and disables an integrity constraint never fails because of rows in the table that violate the integrity constraint. The definition of the constraint is allowed because its rule is not enforced.

17.7.2.2 Enabling Constraints Upon Definition

You can enable an integrity constraint when you define it.

The following CREATE TABLE and ALTER TABLE statements both define and enable integrity constraints:

```
CREATE TABLE emp (
    empno NUMBER(5) CONSTRAINT emp.pk PRIMARY KEY,   . . . ;

ALTER TABLE emp
    ADD CONSTRAINT emp.pk PRIMARY KEY (empno);
```

An ALTER TABLE statement that defines and attempts to enable an integrity constraint can fail because rows of the table violate the integrity constraint. If this case, the statement is rolled back and the constraint definition is not stored and not enabled.

When you enable a UNIQUE or PRIMARY KEY constraint an associated index is created.

> **Note:**
>
> An efficient procedure for enabling a constraint that can make use of parallelism is described in "Efficient Use of Integrity Constraints: A Procedure".

> **See Also:**
>
> "Creating an Index Associated with a Constraint"

17.7.3 Modifying, Renaming, or Dropping Existing Integrity Constraints

You can use the `ALTER TABLE` statement to enable, disable, modify, or drop a constraint. When the database is using a `UNIQUE` or `PRIMARY KEY` index to enforce a constraint, and constraints associated with that index are dropped or disabled, the index is dropped, unless you specify otherwise.

While enabled foreign keys reference a `PRIMARY` or `UNIQUE` key, you cannot disable or drop the `PRIMARY` or `UNIQUE` key constraint or the index.

- Disabling and Enabling Constraints
  You can disable enabled integrity constraints and enable disabled integrity constraints.
- Renaming Constraints
  The `ALTER TABLE...RENAME CONSTRAINT` statement enables you to rename any currently existing constraint for a table. The new constraint name must not conflict with any existing constraint names for a user.
- Dropping Constraints
  You can drop an integrity constraint if the rule that it enforces is no longer true, or if the constraint is no longer needed.

17.7.3.1 Disabling and Enabling Constraints

You can disable enabled integrity constraints and enable disabled integrity constraints.

The following statements disable integrity constraints. The second statement specifies that the associated indexes are to be kept.

```
ALTER TABLE dept
    DISABLE CONSTRAINT dname_ukey;

ALTER TABLE dept
    DISABLE PRIMARY KEY KEEP INDEX,
    DISABLE UNIQUE (dname, loc) KEEP INDEX;
```

The following statements enable novalidate disabled integrity constraints:

```
ALTER TABLE dept
    ENABLE NOVALIDATE CONSTRAINT dname_ukey;

ALTER TABLE dept
    ENABLE NOVALIDATE PRIMARY KEY,
    ENABLE NOVALIDATE UNIQUE (dname, loc);
```

The following statements enable or validate disabled integrity constraints:

```
ALTER TABLE dept
    MODIFY CONSTRAINT dname_key VALIDATE;

ALTER TABLE dept
    MODIFY PRIMARY KEY ENABLE NOVALIDATE;
```

The following statements enable disabled integrity constraints:

```
ALTER TABLE dept
    ENABLE CONSTRAINT dname_ukey;

ALTER TABLE dept
```

```
    ENABLE PRIMARY KEY,
    ENABLE UNIQUE (dname, loc);
```

To disable or drop a `UNIQUE` key or `PRIMARY KEY` constraint and all dependent `FOREIGN KEY` constraints in a single step, use the `CASCADE` option of the `DISABLE` or `DROP` clauses. For example, the following statement disables a `PRIMARY KEY` constraint and any `FOREIGN KEY` constraints that depend on it:

```
ALTER TABLE dept
    DISABLE PRIMARY KEY CASCADE;
```

17.7.3.2 Renaming Constraints

The `ALTER TABLE...RENAME CONSTRAINT` statement enables you to rename any currently existing constraint for a table. The new constraint name must not conflict with any existing constraint names for a user.

The following statement renames the `dname_ukey` constraint for table `dept`:

```
ALTER TABLE dept
    RENAME CONSTRAINT dname_ukey TO dname_unikey;
```

When you rename a constraint, all dependencies on the base table remain valid.

The `RENAME CONSTRAINT` clause provides a means of renaming system generated constraint names.

17.7.3.3 Dropping Constraints

You can drop an integrity constraint if the rule that it enforces is no longer true, or if the constraint is no longer needed.

You can drop the constraint using the `ALTER TABLE` statement with one of the following clauses:

- `DROP PRIMARY KEY`
- `DROP UNIQUE`
- `DROP CONSTRAINT`

The following two statements drop integrity constraints. The second statement keeps the index associated with the `PRIMARY KEY` constraint:

```
ALTER TABLE dept
    DROP UNIQUE (dname, loc);

ALTER TABLE emp
    DROP PRIMARY KEY KEEP INDEX
    DROP CONSTRAINT dept_fkey;
```

If `FOREIGN KEY`s reference a `UNIQUE` or `PRIMARY KEY`, you must include the `CASCADE CONSTRAINTS` clause in the `DROP` statement, or you cannot drop the constraint.

17.7.4 Deferring Constraint Checks

When the database checks a constraint, it signals an error if the constraint is not satisfied. You can defer checking the validity of constraints until the end of a transaction. When you issue the `SET CONSTRAINTS` statement, the `SET CONSTRAINTS` mode lasts for the duration of the transaction, or until another `SET CONSTRAINTS` statement resets the mode.

> **Note:**
>
> - You cannot issue a `SET CONSTRAINT` statement inside a trigger.
> - Deferrable unique and primary keys must use nonunique indexes.

- Set All Constraints Deferred
  When constraints must be deferred for a transaction, you must set all constraints deferred before you actually begin processing any data within the application being used to manipulate the data.
- Check the Commit (Optional)
  You can check for constraint violations before committing by issuing the `SET CONSTRAINTS ALL IMMEDIATE` statement just before issuing the `COMMIT`.

17.7.4.1 Set All Constraints Deferred

When constraints must be deferred for a transaction, you must set all constraints deferred before you actually begin processing any data within the application being used to manipulate the data.

Use the following DML statement to set all deferrable constraints deferred:

```
SET CONSTRAINTS ALL DEFERRED;
```

> **Note:**
>
> The `SET CONSTRAINTS` statement applies only to the current transaction. The defaults specified when you create a constraint remain as long as the constraint exists. The `ALTER SESSION SET CONSTRAINTS` statement applies for the current session only.

17.7.4.2 Check the Commit (Optional)

You can check for constraint violations before committing by issuing the `SET CONSTRAINTS ALL IMMEDIATE` statement just before issuing the `COMMIT`.

If there are any problems with a constraint, then this statement fails and the constraint causing the error is identified. If you commit while constraints are violated, then the transaction is rolled back and you receive an error message.

17.7.5 Reporting Constraint Exceptions

If exceptions exist when a constraint is validated, then an error is returned and the integrity constraint remains novalidated. When a statement is not successfully executed because integrity constraint exceptions exist, the statement is rolled back. If exceptions exist, then you cannot validate the constraint until all exceptions to the constraint are either updated or deleted.

To determine which rows violate the integrity constraint, issue the `ALTER TABLE` statement with the `EXCEPTIONS` option in the `ENABLE` clause. The `EXCEPTIONS` option places the rowid, table owner, table name, and constraint name of all exception rows into a specified table.

You must create an appropriate exceptions report table to accept information from the `EXCEPTIONS` option of the `ENABLE` clause before enabling the constraint. You can create an exception table by executing the `UTLEXCPT.SQL` script or the `UTLEXPT1.SQL` script.

> **Note:**
>
> Your choice of script to execute for creating the `EXCEPTIONS` table depends on the type of table you are analyzing. See the *Oracle Database SQL Language Reference* for more information.

Both of these scripts create a table named `EXCEPTIONS`. You can create additional exceptions tables with different names by modifying and resubmitting the script.

The following statement attempts to validate the `PRIMARY KEY` of the `dept` table, and if exceptions exist, information is inserted into a table named `EXCEPTIONS`:

```
ALTER TABLE dept ENABLE PRIMARY KEY EXCEPTIONS INTO EXCEPTIONS;
```

If duplicate primary key values exist in the `dept` table and the name of the `PRIMARY KEY` constraint on `dept` is `sys_c00610`, then the following query will display those exceptions:

```
SELECT * FROM EXCEPTIONS;
```

The following exceptions are shown:

```
fROWID               OWNER      TABLE_NAME      CONSTRAINT
------------------  ---------  --------------  -----------
AAAAZ9AABAAABvqAAB  SCOTT      DEPT            SYS_C00610
AAAAZ9AABAAABvqAAG  SCOTT      DEPT            SYS_C00610
```

A more informative query would be to join the rows in an exception report table and the master table to list the actual rows that violate a specific constraint, as shown in the following statement and results:

```
SELECT deptno, dname, loc FROM dept, EXCEPTIONS
    WHERE EXCEPTIONS.constraint = 'SYS_C00610'
    AND dept.rowid = EXCEPTIONS.row_id;

DEPTNO     DNAME             LOC
---------- --------------    -----------
10         ACCOUNTING        NEW YORK
10         RESEARCH          DALLAS
```

All rows that violate a constraint must be either updated or deleted from the table containing the constraint. When updating exceptions, you must change the value violating the constraint to a value consistent with the constraint or to a null. After the row in the master table is updated or deleted, the corresponding rows for the exception in the exception report table should be deleted to avoid confusion with later exception reports. The statements that update the master table and the exception report table should be in the same transaction to ensure transaction consistency.

To correct the exceptions in the previous examples, you might issue the following transaction:

```
UPDATE dept SET deptno = 20 WHERE dname = 'RESEARCH';
DELETE FROM EXCEPTIONS WHERE constraint = 'SYS_C00610';
COMMIT;
```

When managing exceptions, the goal is to eliminate all exceptions in your exception report table.

> **Note:**
>
> While you are correcting current exceptions for a table with the constraint disabled, it is possible for other users to issue statements creating new exceptions. You can avoid this by marking the constraint `ENABLE NOVALIDATE` before you start eliminating exceptions.

> **See Also:**
>
> *Oracle Database Reference* for a description of the `EXCEPTIONS` table

17.7.6 Viewing Constraint Information

Oracle Database provides a set of views that enable you to see constraint definitions on tables and to identify columns that are specified in constraints.

View	Description

`DBA_CONSTRAINTS`
`ALL_CONSTRAINTS`
`USER_CONSTRAINTS`
	`DBA` view describes all constraint definitions in the database. `ALL` view describes constraint definitions accessible to current user. `USER` view describes constraint definitions owned by the current user.

`DBA_CONS_COLUMNS`
`ALL_CONS_COLUMNS`
`USER_CONS_COLUMNS`
	`DBA` view describes all columns in the database that are specified in constraints. `ALL` view describes only those columns accessible to current user that are specified in constraints. `USER` view describes only those columns owned by the current user that are specified in constraints.

> **See Also:**
>
> - *Oracle Database Reference* for information about the `*_CONSTRAINTS` views
> - *Oracle Database Reference* for information about the `*_CONS_COLUMNS` views

17.8 Renaming Schema Objects

There are several ways to rename an object.

To rename an object, it must be in your schema. You can rename schema objects in either of the following ways:

- Drop and re-create the object

- Rename the object using the `RENAME` statement
- Rename the object using the `ALTER ... RENAME` statement (for indexes and triggers)

If you drop and re-create an object, all privileges granted for that object are lost. Privileges must be regranted when the object is re-created.

A table, view, sequence, or a private synonym of a table, view, or sequence can be renamed using the `RENAME` statement. When using the `RENAME` statement, integrity constraints, indexes, and grants made for the object are carried forward for the new name. For example, the following statement renames the `sales_staff` view:

```
RENAME sales_staff TO dept_30;
```

> **Note:**
>
> You cannot use `RENAME` for a stored PL/SQL program unit, public synonym, or cluster. To rename such an object, you must drop and re-create it.

Before renaming a schema object, consider the following effects:

- All views and PL/SQL program units dependent on a renamed object become invalid, and must be recompiled before next use.
- All synonyms for a renamed object return an error when used.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for syntax of the `RENAME` statement

17.9 Managing Object Dependencies

Oracle Database provides an automatic mechanism to ensure that a dependent object is always up to date with respect to its referenced objects. You can also manually recompile invalid object.

- About Object Dependencies and Object Invalidation
  Some types of schema objects reference other objects. An object that references another object is called a **dependent object**, and an object being referenced is a **referenced object**. These references are established at compile time, and if the compiler cannot resolve them, the dependent object being compiled is marked *invalid*.
- Manually Recompiling Invalid Objects with DDL
  You can use an `ALTER` statement to manually recompile a single schema object.
- Manually Recompiling Invalid Objects with PL/SQL Package Procedures
  The `RECOMP_SERIAL` procedure recompiles all invalid objects in a specified schema, or all invalid objects in the database if you do not supply the schema name argument. The `RECOMP_PARALLEL` procedure does the same, but in parallel, employing multiple CPUs.

17.9.1 About Object Dependencies and Object Invalidation

Some types of schema objects reference other objects. An object that references another object is called a **dependent object**, and an object being referenced is a **referenced object**. These references are established at compile time, and if the compiler cannot resolve them, the dependent object being compiled is marked *invalid*.

For example, a view contains a query that references tables or other views, and a PL/SQL subprogram might invoke other subprograms and might use static SQL to reference tables or views.

Oracle Database provides an automatic mechanism to ensure that a dependent object is always up to date with respect to its referenced objects. When a dependent object is created, the database tracks dependencies between the dependent object and its referenced objects. When a referenced object is changed in a way that might affect a dependent object, the dependent object is marked invalid. An invalid dependent object must be recompiled against the new definition of a referenced object before the dependent object can be used. Recompilation occurs automatically when the invalid dependent object is referenced.

It is important to be aware of changes that can invalidate schema objects, because invalidation affects applications running on the database. This section describes how objects become invalid, how you can identify invalid objects, and how you can validate invalid objects.

Object Invalidation

In a typical running application, you would not expect to see views or stored procedures become invalid, because applications typically do not change table structures or change view or stored procedure definitions during normal execution. Changes to tables, views, or PL/SQL units typically occur when an application is patched or upgraded using a patch script or ad-hoc DDL statements. Dependent objects might be left invalid after a patch has been applied to change a set of referenced objects.

Use the following query to display the set of invalid objects in the database:

```
SELECT object_name, object_type FROM dba_objects
WHERE status = 'INVALID';
```

The Database Home page in Oracle Enterprise Manager Cloud Control displays an alert when schema objects become invalid.

Object invalidation affects applications in two ways. First, an invalid object must be revalidated before it can be used by an application. Revalidation adds latency to application execution. If the number of invalid objects is large, the added latency on the first execution can be significant. Second, invalidation of a procedure, function or package can cause exceptions in other sessions concurrently executing the procedure, function or package. If a patch is applied when the application is in use in a different session, the session executing the application notices that an object in use has been invalidated and raises one of the following 4 exceptions: ORA-04061, ORA-04064, ORA-04065 or ORA-04068. These exceptions must be remedied by restarting application sessions following a patch.

You can force the database to recompile a schema object using the appropriate SQL statement with the `COMPILE` clause. See "Manually Recompiling Invalid Objects with DDL" for more information.

If you know that there are a large number of invalid objects, use the `UTL_RECOMP` PL/SQL package to perform a mass recompilation. See "Manually Recompiling Invalid Objects with PL/SQL Package Procedures" for details.

The following are some general rules for the invalidation of schema objects:

- Between a referenced object and each of its dependent objects, the database tracks the elements of the referenced object that are involved in the dependency. For example, if a single-table view selects only a subset of columns in a table, only those columns are involved in the dependency. For each dependent of an object, if a change is made to the definition of any element involved in the dependency (including dropping the element), the dependent object is invalidated. Conversely, if changes are made only to definitions of elements that are not involved in the dependency, the dependent object remains valid.

  In many cases, therefore, developers can avoid invalidation of dependent objects and unnecessary extra work for the database if they exercise care when changing schema objects.
- Dependent objects are *cascade invalidated*. If any object becomes invalid for any reason, all of that object's dependent objects are immediately invalidated.
- If you revoke any object privileges on a schema object, dependent objects are cascade invalidated.

> **See Also:**
>
> *Oracle Database Concepts* for more detailed information about schema object dependencies

17.9.2 Manually Recompiling Invalid Objects with DDL

You can use an `ALTER` statement to manually recompile a single schema object.

For example, to recompile package body `Pkg1`, you would execute the following DDL statement:

```
ALTER PACKAGE pkg1 COMPILE REUSE SETTINGS;
```

> **See Also:**
>
> *Oracle Database SQL Language Reference* for syntax and other information about the various `ALTER` statements

17.9.3 Manually Recompiling Invalid Objects with PL/SQL Package Procedures

The `RECOMP_SERIAL` procedure recompiles all invalid objects in a specified schema, or all invalid objects in the database if you do not supply the schema name argument. The `RECOMP_PARALLEL` procedure does the same, but in parallel, employing multiple CPUs.

Following an application upgrade or patch, it is good practice to revalidate invalid objects to avoid application latencies that result from on-demand object revalidation. Oracle provides the `UTL_RECOMP` package to assist in object revalidation.

Examples

Execute the following PL/SQL block to revalidate all invalid objects in the database, in parallel and in dependency order:

```
begin
   utl_recomp.recomp_parallel();
end;
/
```

You can also revalidate individual invalid objects using the package `DBMS_UTILITY`. The following PL/SQL block revalidates the procedure `UPDATE_SALARY` in schema `HR`:

```
begin
   dbms_utility.validate('HR', 'UPDATE_SALARY', namespace=>1);
end;
/
```

The following PL/SQL block revalidates the package body `HR.ACCT_MGMT`:

```
begin
   dbms_utility.validate('HR', 'ACCT_MGMT', namespace=>2);
end;
/
```

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for more information on the `UTL_RECOMP` package
- *Oracle Database PL/SQL Packages and Types Reference* for more information on the `DBMS_UTILITY` package

17.10 Managing Object Name Resolution

Object names referenced in SQL statements can consist of several pieces, separated by periods. Oracle Database performs specific actions to resolve an object name.

The following describes how the database resolves an object name.

1. Oracle Database attempts to qualify the first piece of the name referenced in the SQL statement. For example, in `scott.emp`, `scott` is the first piece. If there is only one piece, the one piece is considered the first piece.
   a. In the current schema, the database searches for an object whose name matches the first piece of the object name. If it does not find such an object, it continues with step 1.b.
   b. The database searches for a public synonym that matches the first piece of the name. If it does not find one, it continues with step 1.c.
   c. The database searches for a schema whose name matches the first piece of the object name. If it finds one, then the schema is the qualified schema, and it continues with step 1.d.

If no schema is found in step 1.c, the object cannot be qualified and the database returns an error.

d. In the qualified schema, the database searches for an object whose name matches the second piece of the object name.

   If the second piece does not correspond to an object in the previously qualified schema or there is not a second piece, then the database returns an error.

2. A schema object has been qualified. Any remaining pieces of the name must match a valid part of the found object. For example, if `scott.emp.deptno` is the name, `scott` is qualified as a schema, `emp` is qualified as a table, and `deptno` must correspond to a column (because `emp` is a table). If `emp` is qualified as a package, `deptno` must correspond to a public constant, variable, procedure, or function of that package.

When global object names are used in a distributed database, either explicitly or indirectly within a synonym, the local database resolves the reference locally. For example, it resolves a synonym to global object name of a remote table. The partially resolved statement is shipped to the remote database, and the remote database completes the resolution of the object as described here.

Because of how the database resolves references, it is possible for an object to depend on the nonexistence of other objects. This situation occurs when the dependent object uses a reference that would be interpreted differently were another object present. For example, assume the following:

- At the current point in time, the `company` schema contains a table named `emp`.
- A `PUBLIC` synonym named `emp` is created for `company.emp` and the `SELECT` privilege for `company.emp` is granted to the `PUBLIC` role.
- The `jward` schema does not contain a table or private synonym named `emp`.
- The user `jward` creates a view in their schema with the following statement:

```
CREATE VIEW dept_salaries AS
     SELECT deptno, MIN(sal), AVG(sal), MAX(sal) FROM emp
     GROUP BY deptno
     ORDER BY deptno;
```

When `jward` creates the `dept_salaries` view, the reference to `emp` is resolved by first looking for `jward.emp` as a table, view, or private synonym, none of which is found, and then as a public synonym named `emp`, which is found. As a result, the database notes that `jward.dept_salaries` depends on the nonexistence of `jward.emp` and on the existence of `public.emp`.

Now assume that `jward` decides to create a new view named `emp` in their schema using the following statement:

```
CREATE VIEW emp AS
     SELECT empno, ename, mgr, deptno
     FROM company.emp;
```

Notice that `jward.emp` does not have the same structure as `company.emp`.

As it attempts to resolve references in object definitions, the database internally makes note of dependencies that the new dependent object has on "nonexistent" objects--schema objects that, if they existed, would change the interpretation of the object's definition. Such dependencies must be noted in case a nonexistent object is later created. If a nonexistent object is created, all dependent objects must be invalidated so that dependent objects can be recompiled and verified and all dependent function-based indexes must be marked unusable.

Therefore, in the previous example, as `jward.emp` is created, `jward.dept_salaries` is invalidated because it depends on `jward.emp`. Then when `jward.dept_salaries` is used, the database attempts to recompile the view. As the database resolves the reference to `emp`, it finds `jward.emp` (`public.emp` is no longer the referenced object). Because `jward.emp` does not have a `sal` column, the database finds errors when replacing the view, leaving it invalid.

In summary, you must manage dependencies on nonexistent objects checked during object resolution in case the nonexistent object is later created.

> **See Also:**
>
> "Schema Objects and Database Links" for information about name resolution in a distributed database

17.11 Switching to a Different Schema

Use an `ALTER SESSION` SQL statement to switch to a different schema.

The following statement sets the schema of the current session to the schema name specified in the statement.

```
ALTER SESSION SET CURRENT_SCHEMA = <schema name>
```

In subsequent SQL statements, Oracle Database uses this schema name as the schema qualifier when the qualifier is omitted. In addition, the database uses the temporary tablespace of the specified schema for sorts, joins, and storage of temporary database objects. The session retains its original privileges and does not acquire any extra privileges by the preceding `ALTER SESSION` statement.

In the following example, provide the password when prompted:

```
CONNECT scott
ALTER SESSION SET CURRENT_SCHEMA = joe;
SELECT * FROM emp;
```

Because `emp` is not schema-qualified, the table name is resolved under schema `joe`. But if `scott` does not have select privilege on table `joe.emp`, then `scott` cannot execute the `SELECT` statement.

17.12 Managing Editions

Application developers who are upgrading their applications using edition-based redefinition may ask you to perform edition-related tasks that require DBA privileges.

- About Editions and Edition-Based Redefinition
  Edition-based redefinition enables you to upgrade an application's database objects while the application is in use, thus minimizing or eliminating down time. This is accomplished by changing (redefining) database objects in a private environment known as an **edition**.
- DBA Tasks for Edition-Based Redefinition
  A user must have the required privileges to perform tasks related to edition-based redefinition.

- Setting the Database Default Edition
  There is always a default edition for the database. This is the edition that a database session initially uses if it does not explicitly indicate an edition when connecting.
- Querying the Database Default Edition
  The database default edition is stored as a database property.
- Setting the Edition Attribute of a Database Service
  You can set the edition attribute of a database service when you create the service, or you can modify an existing database service to set its edition attribute.
- Using an Edition
  To view or modify objects in a particular edition, you must *use* the edition first. You can specify an edition to use when you connect to the database. If you do not specify an edition, then your session starts in the database default edition.
- Editions Data Dictionary Views
  There are several data dictionary views that aid with managing editions.

17.12.1 About Editions and Edition-Based Redefinition

Edition-based redefinition enables you to upgrade an application's database objects while the application is in use, thus minimizing or eliminating down time. This is accomplished by changing (redefining) database objects in a private environment known as an **edition**.

Only when all changes have been made and tested do you make the new version of the application available to users.

> **See Also:**
>
> *Oracle Database Development Guide* for a complete discussion of edition-based redefinition

17.12.2 DBA Tasks for Edition-Based Redefinition

A user must have the required privileges to perform tasks related to edition-based redefinition.

Table 17-1 summarizes the edition-related tasks that require privileges typically granted only to DBAs. Any user that is granted the `DBA` role can perform these tasks.

Table 17-1 DBA Tasks for Edition-Based Redefinition

Task	See
Grant or revoke privileges to create, alter, and drop editions	The `CREATE EDITION` and `DROP EDITION` SQL statements
Enable editions for a schema	*Oracle Database Development Guide*
Set the database default edition	"Setting the Database Default Edition"
Set the edition attribute of a database service	"Setting the Edition Attribute of a Database Service"

17.12.3 Setting the Database Default Edition

There is always a default edition for the database. This is the edition that a database session initially uses if it does not explicitly indicate an edition when connecting.

To set the database default edition:

1. Connect to the database as a user with the `ALTER DATABASE` privilege and `USE` privilege `WITH GRANT OPTION` on the edition.
2. Enter the following statement:

```
ALTER DATABASE DEFAULT EDITION = edition_name;
```

> **See Also:**
>
> "Connecting to the Database with SQL*Plus"

17.12.4 Querying the Database Default Edition

The database default edition is stored as a database property.

To query the database default edition:

1. Connect to the database as any user.
2. Enter the following statement:

```
SELECT PROPERTY_VALUE FROM DATABASE_PROPERTIES WHERE
   PROPERTY_NAME = 'DEFAULT_EDITION';

PROPERTY_VALUE
------------------------------
ORA$BASE
```

> **Note:**
>
> The property name `DEFAULT_EDITION` is case sensitive and must be supplied as upper case.

17.12.5 Setting the Edition Attribute of a Database Service

You can set the edition attribute of a database service when you create the service, or you can modify an existing database service to set its edition attribute.

> **Note:**
>
> The number of database services for an instance has an upper limit. See *Oracle Database Reference* for more information about this limit.

- About Setting the Edition Attribute of a Database Service
  When you set the edition attribute of a service, all subsequent connections that specify the service, such as client connections and `DBMS_SCHEDULER` jobs, use this edition as the initial session edition. However, if a session connection specifies a different edition, then the edition specified in the session connection is used for the session edition.
- Setting the Edition Attribute During Database Service Creation
  You can use the `SRVCTL` utility or the `DBMS_SERVICE` package to set the edition attribute of a database service when you create the service.
- Setting the Edition Attribute of an Existing Database Service
  You can use the `SRVCTL` utility or the `DBMS_SERVICE` package to set the edition attribute of an existing database service.

17.12.5.1 About Setting the Edition Attribute of a Database Service

When you set the edition attribute of a service, all subsequent connections that specify the service, such as client connections and `DBMS_SCHEDULER` jobs, use this edition as the initial session edition. However, if a session connection specifies a different edition, then the edition specified in the session connection is used for the session edition.

To check the edition attribute of a database service, query the `EDITION` column in the `ALL_SERVICES` view or the `DBA_SERVICES` view.

17.12.5.2 Setting the Edition Attribute During Database Service Creation

You can use the `SRVCTL` utility or the `DBMS_SERVICE` package to set the edition attribute of a database service when you create the service.

Follow the instructions in "*Oracle Database SQL Language Reference*" and use the appropriate option for setting the edition attribute for the database service:

- If your single-instance database is being managed by Oracle Restart, use the `SRVCTL` utility to create the database service and specify the `-edition` option to set its edition attribute.

  For the database with the `DB_UNIQUE_NAME` of `dbcrm`, this example creates a new database service named `crmbatch` and sets the edition attribute of the database service to `e2`:

  ```
  srvctl add service -db dbcrm -service crmbatch -edition e2
  ```

- If your single-instance database is not being managed by Oracle Restart, use the `DBMS_SERVICE.CREATE_SERVICE` procedure, and specify the `edition` parameter to set the edition attribute of the database service.

17.12.5.3 Setting the Edition Attribute of an Existing Database Service

You can use the `SRVCTL` utility or the `DBMS_SERVICE` package to set the edition attribute of an existing database service.

To set the edition attribute of an existing database service:

1. Stop the database service.
2. Set the edition attribute of the database service using the appropriate option:
   - If your single-instance database is being managed by Oracle Restart, use the `SRVCTL` utility to modify the database service and specify the `-edition` option to set its edition attribute.

For the database with the `DB_UNIQUE_NAME` of `dbcrm`, this example modifies a database service named `crmbatch` and sets the edition attribute of the service to `e3`:

```
srvctl modify service -db dbcrm -service crmbatch -edition e3
```

- If your single-instance database is not being managed by Oracle Restart, use the `DBMS_SERVICE.MODIFY_SERVICE` procedure, and specify the `edition` parameter to set the edition attribute of the database service. Ensure that the `modify_edition` parameter is set to `TRUE` when you run the `MODIFY_SERVICE` procedure.

3. Start the database service.

> **See Also:**
>
> - Configuring Automatic Restart of an Oracle Database for information managing database services using Oracle Restart
> - *Oracle Database PL/SQL Packages and Types Reference* for information about managing database services using the `DBMS_SERVICE` package

17.12.6 Using an Edition

To view or modify objects in a particular edition, you must *use* the edition first. You can specify an edition to use when you connect to the database. If you do not specify an edition, then your session starts in the database default edition.

To use a different edition, submit the following statement:

```
ALTER SESSION SET EDITION=edition_name;
```

The following statements first set the current edition to `e2` and then to `ora$base`:

```
ALTER SESSION SET EDITION=e2;
...
ALTER SESSION SET EDITION=ora$base;
```

> **See Also:**
>
> - *Oracle Database Development Guide* for more information about using editions, and for instructions for determining the current edition
> - "Connecting to the Database with SQL*Plus"

17.12.7 Editions Data Dictionary Views

There are several data dictionary views that aid with managing editions.

The following table lists three of them. For a complete list, see *Oracle Database Development Guide*.

View	Description
*_EDITIONS	Lists all editions in the database. (Note: USER_EDITIONS does not exist.)
*_OBJECTS	Describes every object in the database that is visible (actual or inherited) in the current edition.
*_OBJECTS_AE	Describes every actual object in the database, across all editions.

17.13 Displaying Information About Schema Objects

Oracle Database provides a PL/SQL package that enables you to determine the DDL that created an object and data dictionary views that you can use to display information about schema objects.

- Using a PL/SQL Package to Display Information About Schema Objects
  The Oracle-supplied PL/SQL package procedure DBMS_METADATA.GET_DDL lets you obtain metadata (in the form of DDL used to create the object) about a schema object.
- Schema Objects Data Dictionary Views
  These views display general information about schema objects.

17.13.1 Using a PL/SQL Package to Display Information About Schema Objects

The Oracle-supplied PL/SQL package procedure DBMS_METADATA.GET_DDL lets you obtain metadata (in the form of DDL used to create the object) about a schema object.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for a description of the DBMS_METADATA package

Example: Using the DBMS_METADATA Package

The DBMS_METADATA package is a powerful tool for obtaining the complete definition of a schema object. It enables you to obtain all of the attributes of an object in one pass. The object is described as DDL that can be used to (re)create it.

In the following statements the GET_DDL function is used to fetch the DDL for all tables in the current schema, filtering out nested tables and overflow segments. The SET_TRANSFORM_PARAM (with the handle value equal to DBMS_METADATA.SESSION_TRANSFORM meaning "for the current session") is used to specify that storage clauses are not to be returned in the SQL DDL. Afterwards, the session-level transform parameters are reset to their defaults. Once set, transform parameter values remain in effect until specifically reset to their defaults.

```
EXECUTE DBMS_METADATA.SET_TRANSFORM_PARAM(
     DBMS_METADATA.SESSION_TRANSFORM,'STORAGE',false);
SELECT DBMS_METADATA.GET_DDL('TABLE',u.table_name)
     FROM USER_ALL_TABLES u
     WHERE u.nested='NO'
     AND (u.iot_type is null or u.iot_type='IOT');
EXECUTE DBMS_METADATA.SET_TRANSFORM_PARAM(
     DBMS_METADATA.SESSION_TRANSFORM,'DEFAULT');
```

The output from DBMS_METADATA.GET_DDL is a LONG data type. When using SQL*Plus, your output may be truncated by default. Issue the following SQL*Plus command before issuing the DBMS_METADATA.GET_DDL statement to ensure that your output is not truncated:

```
SQL> SET LONG 9999
```

17.13.2 Schema Objects Data Dictionary Views

These views display general information about schema objects.

View	Description
DBA_OBJECTS ALL_OBJECTS USER_OBJECTS	DBA view describes all schema objects in the database. ALL view describes objects accessible to current user. USER view describes objects owned by the current user.
DBA_CATALOG ALL_CATALOG USER_CATALOG	List the name, type, and owner (USER view does not display owner) for all tables, views, synonyms, and sequences in the database.
DBA_DEPENDENCIES ALL_DEPENDENCIES USER_DEPENDENCIES	List all dependencies between procedures, packages, functions, package bodies, and triggers, including dependencies on views without any database links.

- Example 1: Displaying Schema Objects By Type
  You can query the USER_OBJECTS view to list all of the objects owned by the user issuing the query.
- Example 2: Displaying Dependencies of Views and Synonyms
  When you create a view or a synonym, the view or synonym is based on its underlying base object. The ALL_DEPENDENCIES, USER_DEPENDENCIES, and DBA_DEPENDENCIES data dictionary views can be used to reveal the dependencies for a view.

17.13.2.1 Example 1: Displaying Schema Objects By Type

You can query the USER_OBJECTS view to list all of the objects owned by the user issuing the query.

The following query lists all of the objects owned by the user issuing the query:

```
SELECT OBJECT_NAME, OBJECT_TYPE
    FROM USER_OBJECTS;
```

The following is the query output:

```
OBJECT_NAME                    OBJECT_TYPE
-----------------------------  -------------------
EMP_DEPT                       CLUSTER
EMP                            TABLE
DEPT                           TABLE
EMP_DEPT_INDEX                 INDEX
PUBLIC_EMP                     SYNONYM
EMP_MGR                        VIEW
```

17.13.2.2 Example 2: Displaying Dependencies of Views and Synonyms

When you create a view or a synonym, the view or synonym is based on its underlying base object. The `ALL_DEPENDENCIES`, `USER_DEPENDENCIES`, and `DBA_DEPENDENCIES` data dictionary views can be used to reveal the dependencies for a view.

The `ALL_SYNONYMS`, `USER_SYNONYMS`, and `DBA_SYNONYMS` data dictionary views can be used to list the base object of a synonym. For example, the following query lists the base objects for the synonyms created by user `jward`:

```
SELECT TABLE_OWNER, TABLE_NAME, SYNONYM_NAME
    FROM DBA_SYNONYMS
    WHERE OWNER = 'JWARD';
```

The following is the query output:

```
TABLE_OWNER             TABLE_NAME   SYNONYM_NAME
----------------------  -----------  ----------------
SCOTT                   DEPT         DEPT
SCOTT                   EMP          EMP
```