15 Creating Compound, DDL, and Event Database Triggers

☑ Describe different types of triggers and their uses

Trigger overview: 9.1

Reasons to use triggers: 9.2

DML triggers: 9.3

Correlation names and pseudorecords: 9.4

Subprograms invoked by triggers: 9.6

Trigger compilation, invalidation, recompilation: 9.7

Trigger exception handling: 9.8

Trigger design: 9.9

Trigger restrictions: 9.10

Trigger firing order: 9.11

Trigger enabling, disabling: 9.12

Trigger changing and debugging: 9.13

Oracle db data transfer and triggers: 9.14

Views for trigger info: 9.16

Overview

- A trigger is like a stored procedure that Oracle Database invokes automatically whenever a specified event occurs
- > Note: The database can detect only system-defined events. You cannot define your own events
- Like a stored procedure, a trigger is a named PL/SQL unit that is stored in the database and can be invoked repeatedly
- Unlike a stored procedure, you can enable and disable a trigger, but you cannot explicitly invoke it
- While a trigger is **enabled**, the database automatically invokes it—that is, the trigger **fires**—whenever its triggering event occurs. While a trigger is **disabled**, it does not fire. By default, a trigger is created in the enabled state
- You create a trigger with the `CREATE TRIGGER` statement. You specify the **triggering event** in terms of **triggering statements** and the item on which they act
- The trigger is said to be **created on** or **defined on** the item, which is either a table, a view, a schema, or the database

- You also specify the **timing point**, which determines whether the trigger fires before or after the triggering statement runs and whether it fires for each row that the triggering statement affects
- If the trigger is created on a table or view, then the triggering event is composed of DML statements, and the trigger is called a **DML trigger**
- A **crossedition trigger** is a DML trigger for use only in edition-based redefinition
- If the trigger is created on a schema or the database, then the triggering event is composed of either DDL or database operation statements, and the trigger is called a **system trigger**
- A **conditional trigger** is a DML or system trigger that has a `WHEN` clause that specifies a SQL condition that the database evaluates for each row that the triggering statement affects
- When a trigger fires, tables that the trigger references might be undergoing changes made by SQL statements in other users' transactions. SQL statements running in triggers follow the same rules that standalone SQL statements do. Specifically:
    - Queries in the trigger see the current read-consistent materialized view of referenced tables and any data changed in the same transaction
    - Updates in the trigger wait for existing data locks to be released before proceeding
- An **INSTEAD OF trigger** is either:
    - A DML trigger created on either a noneditioning view or a nested table column of a noneditioning view
    - A system trigger defined on a `CREATE` statement
- The database fires the `INSTEAD OF` trigger instead of running the triggering statement
- > Note: A trigger is often called by the name of its triggering statement (for example, `DELETE` trigger or `LOGON` trigger), the name of the item on which it is defined (for example, `DATABASE` trigger or `SCHEMA` trigger), or its timing point (for example, `BEFORE` statement trigger or `AFTER` each row trigger)

Reasons to use triggers

- Automatically generate virtual column values
- Log events
- Gather statistics on table access
- Modify table data when DML statements are issued against views
- Enforce referential integrity when child and parent tables are on different nodes of a distributed database
- Publish information about database events, user events, and SQL statements to subscribing applications
- Prevent DML operations on a table after regular business hours
- Prevent invalid transactions
- Enforce complex business or referential integrity rules that you cannot define with constraints
- > Caution: Triggers are not reliable security mechanisms, because they are programmatic and easy to disable. For high-assurance security, use Oracle Database Vault

How triggers and constraints differ

- Both triggers and constraints can constrain data input, but they differ significantly
- A trigger always applies to new data only. For example, a trigger can prevent a DML statement from inserting a `NULL` value into a database column, but the column might contain `NULL` values that were

inserted into the column before the trigger was defined or while the trigger was disabled
- A constraint can apply either to new data only (like a trigger) or to both new and existing data. Constraint behavior depends on constraint state
- Constraints are easier to write and less error-prone than triggers that enforce the same rules. However, triggers can enforce some complex business rules that constraints cannot
- Oracle strongly recommends that you use triggers to constrain data input only in these situations:
    - To enforce referential integrity when child and parent tables are on different nodes of a distributed database
    - To enforce complex business or referential integrity rules that you cannot define with constraints

DML triggers

- A DML trigger is created on either a table or view, and its triggering event is composed of the DML statements `DELETE, INSERT, and UPDATE`
- To create a trigger that fires in response to a `MERGE` statement, create triggers on the `INSERT` and `UPDATE` statements to which the `MERGE` operation decomposes
- A DML trigger is either simple or compound
- A **simple DML trigger** fires at exactly one of these timing points:
    - Before the triggering statement runs (The trigger is called a `BEFORE` statement trigger or statement-level `BEFORE` trigger.)
    - After the triggering statement runs (The trigger is called an `AFTER` statement trigger or statement-level `AFTER` trigger.)
    - Before each row that the triggering statement affects (The trigger is called a `BEFORE` each row trigger or row-level `BEFORE` trigger.)
    - After each row that the triggering statement affects (The trigger is called an `AFTER` each row trigger or row-level `AFTER` trigger.)
- A **compound DML trigger** created on a table or editioning view can fire at one, some, or all of the preceding timing points
    - Compound DML triggers help program an approach where you want the actions that you implement for the various timing points to share common data
- A simple or compound DML trigger that fires at row level can access the data in the row that it is processing
- An **INSTEAD OF DML trigger** is a DML trigger created on either a noneditioning view or a nested table column of a noneditioning view
- Except in an `INSTEAD OF` trigger, a triggering `UPDATE` statement can include a column list. With a column list, the trigger fires only when a specified column is updated. Without a column list, the trigger fires when any column of the associated table is updated

Conditional Predicates for Detecting Triggering DML Statement

- The triggering event of a DML trigger can be composed of multiple triggering statements. When one of them fires the trigger, the trigger can determine which one by using these conditional predicates

    - `INSERTING`: `TRUE` if and only if An `INSERT` statement fired the trigger
    - `UPDATING`: `TRUE` if and only if An `UPDATE` statement fired the trigger
    - `UPDATING ('column')`: `TRUE` if and only if An `UPDATE` statement that affected the specified column fired the trigger
    - `DELETING`: `TRUE` if and only if A `DELETE` statement fired the trigger

- A conditional predicate can appear wherever a `BOOLEAN` expression can appear
- Example:

```
CREATE OR REPLACE TRIGGER t
  BEFORE
    INSERT OR
    UPDATE OF salary, department_id
  ON employees
BEGIN
  CASE
    WHEN INSERTING THEN ...
    WHEN UPDATING('salary') THEN ...
  END CASE;
END;
```

INSTEAD OF DML Triggers

- An `INSTEAD OF` DML trigger is a DML trigger created on a noneditioning view, or on a nested table column of a noneditioning view. The database fires the `INSTEAD OF` trigger instead of running the triggering DML statement
- An `INSTEAD OF` trigger cannot be conditional
- An `INSTEAD OF` trigger is the only way to update a view that is not inherently updatable
- Design the `INSTEAD OF` trigger to determine what operation was intended and do the appropriate DML operations on the underlying tables
- An `INSTEAD OF` trigger is always a row-level trigger
- An `INSTEAD OF` trigger can read `OLD` and `NEW` values, but cannot change them
- An `INSTEAD OF` trigger with the `NESTED TABLE` clause fires only if the triggering statement operates on the elements of the specified nested table column of the view. The trigger fires for each modified nested table element
- Example:

```
CREATE OR REPLACE VIEW order_info AS
   SELECT c.customer_id, c.cust_last_name, c.cust_first_name,
          o.order_id, o.order_date, o.order_status
   FROM customers c, orders o
   WHERE c.customer_id = o.customer_id;

CREATE OR REPLACE TRIGGER order_info_insert
   INSTEAD OF INSERT ON order_info
   BEGIN
     INSERT INTO customers
```

```
    (customer_id, cust_last_name, cust_first_name)
  VALUES (
  :new.customer_id,
  :new.cust_last_name,
  :new.cust_first_name);
INSERT INTO orders (order_id, order_date, customer_id)
VALUES (
  :new.order_id,
  :new.order_date,
  :new.customer_id);
END order_info_insert;
```

Compound DML Triggers

- A compound DML trigger created on a table or editioning view can fire at multiple timing points. Each timing point section has its own executable part and optional exception-handling part, but all of these parts can access a common PL/SQL state
- The common state is established when the triggering statement starts and is destroyed when the triggering statement completes, even when the triggering statement causes an error
- A compound DML trigger created on a noneditioning view is not really compound, because it has only one timing point section
- A compound trigger can be conditional, but not autonomous
- Two common uses of compound triggers are:
    - To accumulate rows destined for a second table so that you can periodically bulk-insert them
    - To avoid the mutating-table error `(ORA-04091)`
- **Compound DML Trigger Structure**
    - The optional declarative part of a compound trigger declares variables and subprograms that all of its timing-point sections can use. When the trigger fires, the declarative part runs before any timing-point sections run. The variables and subprograms exist for the duration of the triggering statement
    - A compound DML trigger created on a noneditioning view is not really compound, because it has only one timing point section. The syntax for creating the simplest compound DML trigger on a noneditioning view is:

```
CREATE trigger FOR dml_event_clause ON view
COMPOUND TRIGGER
INSTEAD OF EACH ROW IS BEGIN
statement;
END INSTEAD OF EACH ROW;
```

- A compound DML trigger created on a table or editioning view has at least one timing-point section out of these: (Timing point: Section)
  - Before the triggering statement runs: `BEFORE STATEMENT`
  - After the triggering statement runs: `AFTER STATEMENT`
  - Before each row that the triggering statement affects: `BEFORE EACH ROW`
  - After each row that the triggering statement affects: `AFTER EACH ROW`
- If the trigger has multiple timing-point sections, they can be in any order, but no timing-point section can be repeated. If a timing-point section is absent, then nothing happens at its timing point
- A compound DML trigger does not have an initialization section, but the `BEFORE STATEMENT` section, which runs before any other timing-point section, can do any necessary initialization
- If a compound DML trigger has neither a `BEFORE STATEMENT` section nor an `AFTER STATEMENT` section, and its triggering statement affects no rows, then the trigger never fires

- **Compound DML Trigger Restrictions**
  - `OLD, NEW, and PARENT` cannot appear in the declarative part, the `BEFORE STATEMENT` section, or the `AFTER STATEMENT` section
  - Only the `BEFORE EACH ROW` section can change the value of `NEW`
  - A timing-point section cannot handle exceptions raised in another timing-point section
  - If a timing-point section includes a `GOTO` statement, the target of the `GOTO` statement must be in the same timing-point section

- **Performance Benefit of Compound DML Triggers**
  - A compound DML trigger has a performance benefit when the triggering statement affects many rows
  - For example, suppose that this statement triggers a compound DML trigger that has all four timing-point sections

```
INSERT INTO Target
  SELECT c1, c2, c3
  FROM Source
  WHERE Source.c1 > 0
```

  - Although the BEFORE EACH ROW and AFTER EACH ROW sections of the trigger run for each row of Source whose column c1 is greater than zero, the BEFORE STATEMENT section runs only before the INSERT statement runs and the AFTER STATEMENT section runs only after the INSERT statement runs
  - A compound DML trigger has a greater performance benefit when it uses bulk SQL

- **Using Compound DML Triggers with Bulk Insertion**

    - A compound DML trigger is useful for accumulating rows destined for a second table so that you can periodically bulk-insert them
    - To get the performance benefit from the compound trigger, you must specify BULK COLLECT INTO in the FORALL statement (otherwise, the FORALL statement does a single-row DML operation multiple times)

- **Using Compound DML Triggers to Avoid Mutating-Table Error**

    - A compound DML trigger is useful for avoiding the mutating-table error (ORA-04091)
    - Example: do the `SELECT INTO` statement in the `BEFORE STATEMENT` timing point so you can use the value in the `AFTER EACH ROW` timing point without having to query at that timing point avoiding mutating table error (mutating table in triggers only happen with row level triggers)

Triggers for Ensuring Referential Integrity

- READ examples

Correlation Names and Pseudorecords

- > Note: This topic applies only to triggers that fire at row level. That is:
  >
  > - Row-level simple DML triggers
  > - Compound DML triggers with row-level timing point sections

- A trigger that fires at row level can access the data in the row that it is processing by using **correlation names**
- The default correlation names are `OLD, NEW, and PARENT`
- To change the correlation names, use the `REFERENCING` clause of the `CREATE TRIGGER` statement
- If the trigger is created on a nested table, then `OLD` and `NEW` refer to the current row of the nested table, and `PARENT` refers to the current row of the parent table
- If the trigger is created on a table or view, then `OLD` and `NEW` refer to the current row of the table or view, and `PARENT` is undefined
- `OLD, NEW, and PARENT` are also called **pseudorecords**, because they have record structure, but are allowed in fewer contexts than records are
- The structure of a pseudorecord is `table_name%ROWTYPE`, where table_name is the name of the table on which the trigger is created (for `OLD` and `NEW`) or the name of the parent table (for `PARENT`)
- In the trigger_body of a simple trigger or the tps_body of a compound trigger, a correlation name is a placeholder for a bind variable. Reference the field of a pseudorecord with this syntax:

  ```
  :pseudorecord_name.field_name
  ```

- In the `WHEN` clause of a conditional trigger, a correlation name is not a placeholder for a bind variable. Therefore, omit the colon in the preceding syntax

- `OLD` and `NEW` field values for triggering statements
    - `INSERT`:
        - old: `NULL`
        - new: post-insert value
    - `UPDATE`:
        - old: pre-update value
        - new: post-update value
    - `DELETE`:
        - old: pre-delete value
        - new: `NULL`
- Restrictions:
    - A pseudorecord cannot appear in a record-level operation. For example, the trigger cannot include this statement: `:NEW := NULL;`
    - A pseudorecord cannot be an actual subprogram parameter. (A pseudorecord field can be an actual subprogram parameter.)
    - The trigger cannot change `OLD` field values. Trying to do so raises `ORA-04085`
    - If the triggering statement is `DELETE`, then the trigger cannot change `NEW` field values. Trying to do so raises `ORA-04084`
    - An `AFTER` trigger cannot change `NEW` field values, because the triggering statement runs before the trigger fires. Trying to do so raises `ORA-04084`
- A `BEFORE` trigger can change `NEW` field values before a triggering `INSERT` or `UPDATE` statement puts them in the table
- If a statement triggers both a `BEFORE` trigger and an `AFTER` trigger, and the `BEFORE` trigger changes a `NEW` field value, then the `AFTER` trigger "sees" that change

OBJECT_VALUE Pseudocolumn

- A DML trigger on an object table can reference the SQL pseudocolumn `OBJECT_VALUE`, which returns system-generated names for the columns of the object table
- The trigger can also invoke a PL/SQL subprogram that has a formal `IN` parameter whose data type is `OBJECT_VALUE`

Subprograms Invoked by Triggers

- Triggers can invoke subprograms written in PL/SQL, C, and Java
- A subprogram invoked by a trigger cannot run transaction control statements, because the subprogram runs in the context of the trigger body
- If a trigger invokes an invoker rights (IR) subprogram, then the user who created the trigger, not the user who ran the triggering statement, is considered to be the current user
- If a trigger invokes a remote subprogram, and a time stamp or signature mismatch is found during execution of the trigger, then the remote subprogram does not run and the trigger is invalidated

Trigger Compilation, Invalidation, and Recompilation

- The `CREATE TRIGGER` statement compiles the trigger and stores its code in the database

- If a compilation error occurs, the trigger is still created, but its triggering statement fails, except in these cases:
  - The trigger was created in the disabled state
  - The triggering event is `AFTER STARTUP ON DATABASE`
  - The triggering event is either `AFTER LOGON ON DATABASE` or `AFTER LOGON ON SCHEMA`, and someone logs on as `SYSTEM`
- To see trigger compilation errors, either use the `SHOW ERRORS` command in SQL*Plus or Enterprise Manager, or query the static data dictionary view `*_ERRORS`
- If a trigger does not compile successfully, then its exception handler cannot run
- If a trigger references another object, such as a subprogram or package, and that object is modified or dropped, then the trigger becomes invalid. The next time the triggering event occurs, the compiler tries to revalidate the trigger
- To recompile a trigger manually, use the ALTER TRIGGER statement

Exception Handling in Triggers

- In most cases, if a trigger runs a statement that raises an exception, and the exception is not handled by an exception handler, then the database rolls back the effects of both the trigger and its triggering statement
- In the following cases, the database rolls back only the effects of the trigger, not the effects of the triggering statement (and logs the error in trace files and the alert log):
  - The triggering event is either `AFTER STARTUP ON DATABASE` or `BEFORE SHUTDOWN ON DATABASE`
  - The triggering event is `AFTER LOGON ON DATABASE` and the user has the `ADMINISTER DATABASE TRIGGER` privilege
  - The triggering event is `AFTER LOGON ON SCHEMA` and the user either owns the schema or has the `ALTER ANY TRIGGER` privilege
- In the case of a compound DML trigger, the database rolls back only the effects of the triggering statement, not the effects of the trigger. However, variables declared in the trigger are re-initialized, and any values computed before the triggering statement was rolled back are lost
- > Note: Triggers that enforce complex security authorizations or constraints typically raise user-defined exceptions

Remote Exception Handling

- A trigger that accesses a remote database can do remote exception handling only if the remote database is available. If the remote database is unavailable when the local database must compile the trigger, then the local database cannot validate the statement that accesses the remote database, and the compilation fails. If the trigger cannot be compiled, then its exception handler cannot run
- workaround: Put the remote `INSERT` statement and exception handler in a stored subprogram and have the trigger invoke the stored subprogram. The subprogram is stored in the local database in compiled form, with a validated statement for accessing the remote database. Therefore, when the remote `INSERT` statement fails because the remote database is unavailable, the exception handler in the subprogram can handle it

Trigger Design Guidelines

- Use triggers to ensure that whenever a specific event occurs, any necessary actions are done (regardless of which user or application issues the triggering statement). For example, use a trigger to ensure that whenever anyone updates a table, its log file is updated
- Do not create triggers that duplicate database features. For example, do not create a trigger to reject invalid data if you can do the same with constraints
- Do not create triggers that depend on the order in which a SQL statement processes rows (which can vary). For example, do not assign a value to a global package variable in a row trigger if the current value of the variable depends on the row being processed by the row trigger. If a trigger updates global package variables, initialize those variables in a `BEFORE` statement trigger
- Use `BEFORE` row triggers to modify the row before writing the row data to disk
- Use `AFTER` row triggers to obtain the row ID and use it in operations. An `AFTER` row trigger fires when the triggering statement results in `ORA-02292`
    - > Note: `AFTER` row triggers are slightly more efficient than `BEFORE` row triggers. With `BEFORE` row triggers, affected data blocks are read first for the trigger and then for the triggering statement. With `AFTER` row triggers, affected data blocks are read only for the trigger
- If the triggering statement of a `BEFORE` statement trigger is an `UPDATE` or `DELETE` statement that conflicts with an `UPDATE` statement that is running, then the database does a transparent `ROLLBACK` to `SAVEPOINT` and restarts the triggering statement. The database can do this many times before the triggering statement completes successfully. Each time the database restarts the triggering statement, the trigger fires. The `ROLLBACK` to `SAVEPOINT` does not undo changes to package variables that the trigger references. To detect this situation, include a counter variable in the package
- Do not create recursive triggers. For example, do not create an `AFTER UPDATE` trigger that issues an `UPDATE` statement on the table on which the trigger is defined. The trigger fires recursively until it runs out of memory
- If you create a trigger that includes a statement that accesses a remote database, then put the exception handler for that statement in a stored subprogram and invoke the subprogram from the trigger
- Use `DATABASE` triggers judiciously. They fire every time any database user initiates a triggering event
- If a trigger runs the following statement, the statement returns the owner of the trigger, not the user who is updating the table: `SELECT Username FROM USER_USERS;`
- Only committed triggers fire. A trigger is committed, implicitly, after the CREATE TRIGGER statement that creates it succeeds. Therefore, the following statement cannot fire the trigger that it creates:

```
CREATE OR REPLACE TRIGGER my_trigger
  AFTER CREATE ON DATABASE
BEGIN
  NULL;
```

```
END;
/
```

- To allow the modular installation of applications that have triggers on the same tables, create multiple triggers of the same type, rather than a single trigger that runs a sequence of operations. Each trigger sees the changes made by the previously fired triggers. Each trigger can see `OLD` and `NEW` values

Trigger Restrictions

- Trigger Size Restriction
  - The size of the trigger cannot exceed 32K
  - If the logic for your trigger requires much more than 60 lines of PL/SQL source text, then put most of the source text in a stored subprogram and invoke the subprogram from the trigger
- Trigger LONG and LONG RAW Data Type Restrictions
  - A trigger cannot declare a variable of the `LONG` or `LONG RAW` data type
  - A SQL statement in a trigger can reference a `LONG` or `LONG RAW` column only if the column data can be converted to the data type `CHAR` or `VARCHAR2`
  - A trigger cannot use the correlation name `NEW` or `PARENT` with a `LONG` or `LONG RAW` column
- Mutating-Table Restriction
  - > Note: This topic applies only to row-level simple DML triggers
  - A mutating table is a table that is being modified by a DML statement (possibly by the effects of a `DELETE CASCADE` constraint). (A view being modified by an `INSTEAD OF` trigger is not considered to be mutating.)
  - The mutating-table restriction prevents the trigger from querying or modifying the table that the triggering statement is modifying.
  - When a row-level trigger encounters a mutating table, `ORA-04091` occurs, the effects of the trigger and triggering statement are rolled back, and control returns to the user or application that issued the triggering statement
  - > Caution: Oracle Database does not enforce the mutating-table restriction for a trigger that accesses remote nodes. Similarly, the database does not enforce the mutating-table restriction for tables in the same database that are connected by loop-back database links.
  - If you must use a trigger to update a mutating table, you can avoid the mutating-table error in either of these ways:
    - Use a compound DML trigger
    - Use a temporary table. For example, instead of using one `AFTER each row` trigger that updates the mutating table, use two triggers—an `AFTER each row` trigger that updates the temporary table and an `AFTER statement` trigger that updates the mutating table with the values from the temporary table
    - Use autonomous transactions
- Only an autonomous trigger can run TCL or DDL statements
- A trigger cannot invoke a subprogram that runs transaction control statements, because the subprogram runs in the context of the trigger body
- A trigger cannot access a `SERIALLY_REUSABLE` package

Order in Which Triggers Fire

- If two or more triggers with different timing points are defined for the same statement on the same table, then they fire in this order:
    1. All `BEFORE STATEMENT` triggers
    2. All `BEFORE EACH ROW` triggers
    3. All `AFTER EACH ROW` triggers
    4. All `AFTER STATEMENT` triggers
- If it is practical, replace the set of individual triggers with different timing points with a single compound trigger that explicitly codes the actions in the order you intend
- If you are creating two or more triggers with the same timing point, and the order in which they fire is important, then you can control their firing order using the `FOLLOWS` and `PRECEDES` clauses
- If multiple compound triggers are created on a table, then:
    - All `BEFORE STATEMENT` sections run at the `BEFORE STATEMENT` timing point, `BEFORE EACH ROW` sections run at the `BEFORE EACH ROW` timing point, and so forth
        - If trigger execution order was specified using the `FOLLOWS` clause, then the `FOLLOWS` clause determines the order of execution of compound trigger sections
        - If `FOLLOWS` is specified for some but not all triggers, then the order of execution of triggers is guaranteed only for those that are related using the `FOLLOWS` clause
    - All `AFTER STATEMENT` sections run at the `AFTER STATEMENT` timing point, `AFTER EACH ROW` sections run at the `AFTER EACH ROW` timing point, and so forth
        - If trigger execution order was specified using the `PRECEDES` clause, then the `PRECEDES` clause determines the order of execution of compound trigger sections
        - If `PRECEDES` is specified for some but not all triggers, then the order of execution of triggers is guaranteed only for those that are related using the `PRECEDES` clause
    - > Note: PRECEDES applies only to reverse crossedition triggers
- The firing of compound triggers can be interleaved with the firing of simple triggers
- When one trigger causes another trigger to fire, the triggers are said to be **cascading**
    - The database allows up to 32 triggers to cascade simultaneously
    - To limit the number of trigger cascades, use the initialization parameter `OPEN_CURSORS`, because a cursor opens every time a trigger fires

Trigger Enabling and Disabling

- By default, the `CREATE TRIGGER` statement creates a trigger in the enabled state
- To create a trigger in the disabled state, specify `DISABLE`. Creating a trigger in the disabled state lets you ensure that it compiles without errors before you enable it
- Some reasons to temporarily disable a trigger are:
    - The trigger refers to an unavailable object
    - You must do a large data load, and you want it to proceed quickly without firing triggers
    - You are reloading data
- To enable or disable a single trigger, use this statement:

```
ALTER TRIGGER [schema.]trigger_name { ENABLE | DISABLE };
```

- To enable or disable all triggers in all editions created on a specific table, use this statement:

```
ALTER TABLE table_name { ENABLE | DISABLE } ALL TRIGGERS;
```

Trigger Changing and Debugging

- To change a trigger, you must either replace or re-create it. (The `ALTER TRIGGER` statement only enables, disables, compiles, or renames a trigger.)
- To replace a trigger, use the `CREATE TRIGGER` statement with the `OR REPLACE` clause
- To re-create a trigger, first drop it with the `DROP TRIGGER` statement and then create it again with the `CREATE TRIGGER` statement
- To debug a trigger, you can use the facilities available for stored subprograms

Triggers and Oracle Database Data Transfer Utilities

- The Oracle database utilities that transfer data to your database, possibly firing triggers, are:
  - SQL*Loader (sqlldr)
  - Data Pump Import (impdp)
  - Original Import (imp)

Views for Information About Triggers

- The `*_TRIGGERS` static data dictionary views reveal information about triggers

✅Create triggers on DDL statements

System triggers: 9.5

- A **system trigger** is created on either a schema or the database
- Its triggering event is composed of either DDL statements or database operation statements
- A system trigger fires at exactly one of these timing points:
  - Before the triggering statement runs (The trigger is called a `BEFORE statement` trigger or `statement-level BEFORE` trigger.)
  - After the triggering statement runs (The trigger is called a `AFTER statement` trigger or `statement-level AFTER` trigger.)
  - Instead of the triggering `CREATE` statement (The trigger is called an `INSTEAD OF CREATE` trigger.)

SCHEMA Triggers

- A **SCHEMA trigger** is created on a schema and fires whenever the user who owns it is the current user and initiates the triggering event
- Suppose that both user1 and user2 own schema triggers, and user1 invokes a DR unit owned by user2. Inside the DR unit, user2 is the current user. Therefore, if the DR unit initiates the triggering event of a

schema trigger that user2 owns, then that trigger fires. However, if the DR unit initiates the triggering event of a schema trigger that user1 owns, then that trigger does not fire
- Example: create a `BEFORE` statement trigger on the sample schema HR. When a user **connected as HR** tries to drop a database object, the database fires the trigger before dropping the object

DATABASE Triggers

- A **DATABASE trigger** is created on the database and fires whenever any database user initiates the triggering event
- > Note: An `AFTER SERVERERROR` trigger fires only if Oracle relational database management system (RDBMS) determines that it is safe to fire error triggers

INSTEAD OF CREATE Triggers

- An `INSTEAD OF CREATE` trigger is a `SCHEMA` trigger whose triggering event is a `CREATE` statement
- The database fires the trigger instead of executing its triggering statement
- Example:

```
CREATE OR REPLACE TRIGGER t
  INSTEAD OF CREATE ON SCHEMA
  BEGIN
    EXECUTE IMMEDIATE 'CREATE TABLE T (n NUMBER, m NUMBER)';
  END;
/
```

☑Create triggers on system events

CREATE TRIGGER statement

ALTER TRIGGER statement

DROP TRIGGER statement

CREATE TRIGGER statement

- The `CREATE TRIGGER` statement creates or replaces a database trigger, which is either of these:
  - A stored PL/SQL block associated with a table, a view, a schema, or the database
  - An anonymous PL/SQL block or an invocation of a procedure implemented in PL/SQL or Java
- The database automatically runs a trigger when specified conditions occur
- Prerequisites
  - To create a trigger in your schema on a table in your schema or on your schema (`SCHEMA`), you must have the `CREATE TRIGGER` system privilege
  - To create a trigger in any schema on a table in any schema, or on another user's schema (`schema.SCHEMA`), you must have the `CREATE ANY TRIGGER` system privilege
  - In addition to the preceding privileges, to create a trigger on `DATABASE`, you must have the `ADMINISTER DATABASE TRIGGER` system privilege

    - To create a trigger on a pluggable database (PDB), you must be connected to that PDB and have the `ADMINISTER DATABASE TRIGGER` system privilege
    - In addition to the preceding privileges, to create a crossedition trigger, you must be enabled for editions
    - If the trigger issues SQL statements or invokes procedures or functions, then the owner of the trigger must have the privileges necessary to perform these operations. These privileges must be granted directly to the owner rather than acquired through roles
- syntax and semantics:
    - **create_trigger**

```
CREATE [OR REPLACE] [EDITIONABLE | NONEDITIONABLE] TRIGGER
plsql_trigger_source
```

    - **plsql_trigger_source**

```
[schema.] trigger_name [sharing_clause] [default_collation_clause]
{simple_dml_trigger | instead_of_dml_trigger | compound_dml_trigger |
system_trigger}
```

        - Triggers in the same schema cannot have the same names
        - Triggers can have the same names as other schema objects—for example, a table and a trigger can have the same name—however, to avoid confusion, this is not recommended
        - If a trigger produces compilation errors, then it is still created, but it fails on execution
        - A trigger that fails on execution effectively blocks all triggering DML statements until it is disabled, replaced by a version without compilation errors, or dropped
        - You can see the associated compiler error messages with the SQL*Plus command `SHOW ERRORS`
        - > If you create a trigger on a base table of a materialized view, then you must ensure that the trigger does not fire during a refresh of the materialized view. During refresh, the `DBMS_MVIEW` procedure `I_AM_A_REFRESH` returns `TRUE`
    - **simple_dml_trigger**

```
{BEFORE | AFTER} dml_event_clause [referencing_clause] [FOR EACH ROW]
[trigger_edition_clause]
[trigger_ordering_clause] [ENABLE | DISABLE] [WHEN (condition)]
trigger_body
```

        - You cannot specify a `BEFORE trigger` or `AFTER trigger` on a view unless it is an editioning view
        - In a `BEFORE statement trigger`, the trigger body cannot read `:NEW` or `:OLD`. (In a `BEFORE row trigger`, the trigger body can read and write the `:OLD` and `:NEW` fields.)

- In an `AFTER statement trigger`, the trigger body cannot read `:NEW` or `:OLD`. (In an `AFTER row trigger`, the trigger body can read but not write the `:OLD` and `:NEW` fields.)
- `FOR EACH ROW`:
  - Creates the trigger as a row trigger. The database fires a row trigger for each row that is affected by the triggering statement and meets the optional trigger constraint defined in the `WHEN` condition
  - If you omit this clause, then the trigger is a statement trigger. The database fires a statement trigger only when the triggering statement is issued if the optional trigger constraint is met
- `WHEN (condition)`:
  - Specifies a SQL condition that the database evaluates for each row that the triggering statement affects. If the value of condition is `TRUE` for an affected row, then trigger_body runs for that row; otherwise, trigger_body does not run for that row
  - The triggering statement runs regardless of the value of condition
  - The condition can contain correlation names. In condition, do not put a colon [:] before the correlation name `NEW`, `OLD`, or `PARENT` (in this context, it is not a placeholder for a bind variable)
  - If you specify this clause, then you must also specify `FOR EACH ROW`
  - The condition cannot include a subquery or a PL/SQL expression

- **instead_of_dml_trigger**

```
INSTEAD OF {DELETE | INSERT | UPDATE} [OR {...}] ON [NESTED TABLE
nested_table_column OF]
[schema.] noneditioning_view [referencing_clause] [FOR EACH ROW]
[trigger_edition_clause]
[trigger_ordering_clause] [ENABLE | DISABLE] trigger_body
```

- An `INSTEAD OF trigger` can read the `:OLD` and `:NEW` values, but cannot change them
- If the view is inherently updatable and has `INSTEAD OF triggers`, the triggers take precedence: The database fires the triggers instead of performing DML on the view
- If the view belongs to a hierarchy, then the subviews do not inherit the trigger
- The `WITH CHECK OPTION` for views is not enforced when inserts or updates to the view are done using `INSTEAD OF triggers`. The `INSTEAD OF trigger body` must enforce the check
- The database fine-grained access control lets you define row-level security policies on views. These policies enforce specified rules in response to DML operations. If an `INSTEAD OF trigger` is also defined on the view, then the database does not enforce the row-level security policies, because the database fires the `INSTEAD OF trigger` instead of running the DML on the view
- If the trigger is created on a noneditioning view, then `DELETE, INSERT, UPDATE` causes the database to fire the trigger whenever a `DELETE, INSERT, UPDATE` statement {removes | adds | changes} {a row from the table | a row to the table | a value in a column of the table} on which the noneditioning view is defined

- **compound_dml_trigger**

```
FOR dml_event_clause [referencing_clause] [trigger_edition_clause]
[trigger_ordering_clause]
[ENABLE | DISABLE] [WHEN (condition)] compound_trigger_block
```

  - `WHEN (condition)`:
    - If you specify this clause, then you must also specify at least one of these timing points:
      - `BEFORE EACH ROW`
      - `AFTER EACH ROW`
      - `INSTEAD OF EACH ROW`
    - The condition cannot include a subquery or a PL/SQL expression

- **system_trigger**

```
{BEFORE | AFTER | INSTEAD OF} {ddl_event [OR ...] | database_event [OR
...]} ON
{[schema.] SCHEMA | [PLUGGABLE] DATABASE} [trigger_ordering_clause]
[ENABLE | DISABLE] trigger_body
```

  - `INSTEAD OF`:
    - Creates an `INSTEAD OF` trigger
    - The triggering event must be a `CREATE` statement
    - You can create at most one `INSTEAD OF` DDL trigger. For example, you can create an `INSTEAD OF` trigger on either the database or schema, but not on both the database and schema
  - ddl_event
    - You can create triggers for these events on `DATABASE` or `SCHEMA` unless otherwise noted
    - You can create `BEFORE` and `AFTER` triggers for any of these events, but you can create `INSTEAD OF` triggers only for `CREATE` events
    - The database fires the trigger in the existing user transaction
    - > Note: Some objects are created, altered, and dropped using PL/SQL APIs (for example, scheduler jobs are maintained by subprograms in the `DBMS_SCHEDULER` package). Such PL/SQL subprograms do not fire DDL triggers
    - valid ddl_events:
      - `ALTER, ANALYZE, ASSOCIATE STATISTICS, AUDIT, COMMENT, CREATE, DISASSOCIATE STATISTICS, DROP, GRANT, NOAUDIT, RENAME, REVOKE, TRUNCATE, DDL`
  - database_event

        - You can create triggers for these events on either `DATABASE` or `SCHEMA` unless otherwise noted
        - For each of these triggering events, the database opens an autonomous transaction scope, fires the trigger, and commits any separate transaction (regardless of any existing user transaction)
        - valid database_events:
            - `AFTER STARTUP`: Causes the database to fire the trigger whenever the database is opened. This event is valid only with `DATABASE`, not with `SCHEMA`
            - `BEFORE SHUTDOWN`: Causes the database to fire the trigger whenever an instance of the database is shut down. This event is valid only with `DATABASE`, not with `SCHEMA`
            - `AFTER DB_ROLE_CHANGE`
            - `AFTER SERVERERROR`: Causes the database to fire the trigger whenever both of these conditions are true:
                - A server error message is logged
                - Oracle relational database management system (RDBMS) determines that it is safe to fire error triggers. Unsafe examples: RDBMS is starting up, A critical error has occurred
            - `AFTER LOGON`
            - `BEFORE LOGOFF`
            - `AFTER SUSPEND`
            - `AFTER CLONE`
                - Can be specified only if `PLUGGABLE DATABASE` is specified
            - `BEFORE UNPLUG`
                - Can be specified only if `PLUGGABLE DATABASE` is specified
            - `[ BEFORE | AFTER ] SET CONTAINER`
    - `WHEN (condition)`:
        - You cannot specify this clause for a `STARTUP`, `SHUTDOWN`, or `DB_ROLE_CHANGE` trigger
        - If you specify this clause for a `SERVERERROR` trigger, then condition must be `ERRNO = error_code`
        - The condition cannot include a subquery, a PL/SQL expression (for example, an invocation of a user-defined function), or a correlation name
    - trigger_body
        - The trigger body cannot specify either `:NEW` or `:OLD`

- **dml_event_clause**

```
{DELETE | INSERT | UPDATE [OF column [,...]]} [OR ...] ON [schema.]
{table | view}
```

    - You cannot create a trigger on a table in the schema `SYS`

- **referencing_clause**

```
REFERENCING {OLD [AS] old_name | NEW [AS] new_name | PARENT [AS]
parent_name}
```

- Specifies correlation names, which refer to old, new, and parent values of the current row. Defaults: OLD, NEW, and PARENT
- If your trigger is associated with a table named `OLD`, `NEW`, or `PARENT`, then use this clause to specify different correlation names to avoid confusion between the table names and the correlation names
- The referencing_clause is not valid if trigger_body is `CALL` routine
- DML row-level triggers cannot reference fields of `OLD/NEW/PARENT` pseudorecords (correlation names) that correspond to columns with declared collation other than `USING_NLS_COMP`

- **trigger_ordering_clause**

```
{FOLLOWS | PRECEDES} [schema.] trigger [,...]
```

- `FOLLOWS | PRECEDES` specifies the relative firing of triggers that have the same timing point
- It is especially useful when creating crossedition triggers, which must fire in a specific order to achieve their purpose
- Use `FOLLOWS` to indicate that the trigger being created must fire after the specified triggers. You can specify `FOLLOWS` for a conventional trigger or for a forward crossedition trigger
- Use `PRECEDES` to indicate that the trigger being created must fire before the specified triggers. You can specify `PRECEDES` only for a reverse crossedition trigger
- The specified triggers must exist, and they must have been successfully compiled. They need not be enabled
- If you are creating a noncrossedition trigger, then the specified triggers must be all of the following:
  - Noncrossedition triggers
  - Defined on the same table as the trigger being created
  - Visible in the same edition as the trigger being created
- In the following definitions, A, B, C, and D are either noncrossedition triggers or forward crossedition triggers:
  - If B specifies A in its `FOLLOWS` clause, then B directly follows A
  - If C directly follows B, and B directly follows A, then C indirectly follows A
  - If D directly follows C, and C indirectly follows A, then D indirectly follows A
  - If B directly or indirectly follows A, then B explicitly follows A (that is, the firing order of B and A is explicitly specified by one or more `FOLLOWS` clauses)

- **trigger_body**

```
{plsql_block | CALL routine_clause}
```

- The PL/SQL block or `CALL` subprogram that the database runs to fire the trigger
- A `CALL` subprogram is either a PL/SQL subprogram or a Java subprogram in a PL/SQL wrapper
- If trigger_body is a PL/SQL block and it contains errors, then the `CREATE [OR REPLACE]` statement fails
- The declare_section cannot declare variables of the data type `LONG` or `LONG RAW`

- **compound_trigger_block**

```
COMPOUND TRIGGER [declare_section] timing_point_section
[timing_point_section ...] END [trigger] ;
```

- If the trigger is created on a noneditioning view, then compound_trigger_block must have only the `INSTEAD OF EACH ROW` section
- If the trigger is created on a table or editioning view, then timing point sections can be in any order, but no section can be repeated. The compound_trigger_block cannot have an `INSTEAD OF EACH ROW` section
- The declare_section of compound_trigger_block cannot include `PRAGMA AUTONOMOUS_TRANSACTION`

- **timing_point_section**

```
timing_point IS BEGIN tps_body END timing_point ;
```

- **timing_point**

```
{BEFORE STATEMENT | BEFORE EACH ROW | AFTER STATEMENT | AFTER EACH ROW
| INSTEAD OF EACH ROW}
```

- `BEFORE STATEMENT`
  - This section cannot specify `:NEW` or `:OLD`
- `BEFORE EACH ROW`
  - The trigger fires before each affected row is changed
  - This section can read and write the :OLD and :NEW fields
- `AFTER EACH ROW`
  - The trigger fires after each affected row is changed
  - This section can read but not write the `:OLD` and `:NEW` fields
- `AFTER STATEMENT`
  - This section cannot specify `:NEW` or `:OLD`

        - `INSTEAD OF EACH ROW`
            - Specifies the `INSTEAD OF EACH ROW` section (the only timing point section) of a compound_dml_trigger on a noneditioning view
            - The database runs tps_body instead of running the triggering DML statement
            - This section can appear only in a compound_dml_trigger on a noneditioning view
            - This section can read but not write the `:OLD` and `:NEW` values

    - **tps_body**

```
statements [EXCEPTION exception_handlers]
```

ALTER TRIGGER statement

- The `ALTER TRIGGER` statement enables, disables, compiles, or renames a database trigger
- Prerequisites
    - If the trigger is in the `SYS` schema, you must be connected as `SYSDBA`. Otherwise, the trigger must be in your schema or you must have `ALTER ANY TRIGGER` system privilege
    - In addition, to alter a trigger on `DATABASE`, you must have the `ADMINISTER DATABASE TRIGGER` system privilege
- syntax and semantics:

    - alter_trigger

```
ALTER TRIGGER [schema.] trigger_name {trigger_compile_clause | {ENABLE
| DISABLE} |
RENAME TO new_name | {EDITIONABLE | NONEDITIONABLE}} ;
```

        - `RENAME`: Renames the trigger without changing its state
        - You cannot specify `NONEDITIONABLE` for a crossedition trigger

    - trigger_compile_clause

```
COMPILE [DEBUG] [compiler_parameters_clause] [REUSE SETTINGS]
```

DROP TRIGGER statement

- The `DROP TRIGGER` statement drops a database trigger from the database
- Prerequisites
    - The trigger must be in your schema or you must have the `DROP ANY TRIGGER` system privilege
    - To drop a trigger on `DATABASE` in another user's schema, you must also have the `ADMINISTER DATABASE TRIGGER` system privilege
- syntax and semantics:

    - drop_trigger

```
DROP TRIGGER [schema.] trigger ;
```