Triggers in Oracle Database

In the Oracle Database Management System realm, a **trigger** is a specialized piece of PL/SQL code. It's designed to automatically execute or fire when certain events occur within the database.

Unlike a stored procedure, which needs to be explicitly invoked, a trigger operates differently. It's intrinsically tied to the event associated with it and only executes in response to that specific event. This automatic execution makes triggers a powerful tool for maintaining the integrity of the data within the database.

Triggers can fire before or after an event and on either row-level or statement-level actions. This flexibility allows database administrators to enforce complex business rules and relationships between tables.

In conclusion, triggers are an essential component of the Oracle Database Management System, providing automated, event-driven actions that help maintain data consistency and integrity.

A function block that is automatically executed by some event in an SQL database.

Events

- A DML Statement
- A DDL Statement
- System Event: shutdown or startup of database
- User Event: Login / Logout / Do something with the database

Types of triggers

a. DML Triggers
    i. Insert Trigger
    ii. Update Trigger
    iii. Delete Trigger
b. DDL Triggers
    i. Create type trigger

   ii. Alter type trigger

c. System Event Triggers
   i. Log trigger
   ii. Time Trigger

d. Instead of Triggers
   i. The trigger for non-updatable view
   ii. The trigger for invalid query handle

e. Compound Triggers
   i. Role-based triggers
   ii. Multi-purpose triggers

Trigger Example 1

```
CREATE OR REPLACE TRIGGER SAMPLE_TRIGGER
BEFORE INSERT ON SAMPLE_TABLE
FOR EACH ROW
ENABLE
DECLARE
    v_count NUMBER;
BEGIN
    SELECT COUNT(*) INTO v_count FROM SAMPLE_TABLE;
    IF v_count > 10 THEN
        RAISE_APPLICATION_ERROR(-20001, 'Cannot insert more than 10
records');
    END IF;
END;
/
```

This code is creating a trigger named `SAMPLE_TRIGGER` in Oracle Database. Here's what each part does:

1. `CREATE OR REPLACE TRIGGER SAMPLE_TRIGGER`: This line is creating (or replacing, if it already exists) a trigger named `SAMPLE_TRIGGER`.
2. `BEFORE INSERT ON SAMPLE_TABLE`: This specifies when the trigger should be executed. In this case, it's set to run before an `INSERT` operation on `SAMPLE_TABLE`.

3. `FOR EACH ROW`: This means the trigger will execute for each row being inserted.
4. `ENABLE`: This keyword ensures that the trigger is enabled and can be fired when the specified event occurs.
5. `DECLARE v_count NUMBER;`: This line declares a variable `v_count` of type `NUMBER`.
6. `BEGIN ... END;`: These keywords enclose the main logic of the trigger.
7. `SELECT COUNT(*) INTO v_count FROM SAMPLE_TABLE;`: This line counts the number of rows in `SAMPLE_TABLE` and stores the result into `v_count`.
8. `IF v_count > 10 THEN RAISE_APPLICATION_ERROR(-20001, 'Cannot insert more than 10 records'); END IF;`: This is a conditional statement. If `v_count` is greater than 10, it raises an application error with the message 'Cannot insert more than 10 records'. This effectively limits the number of rows in `SAMPLE_TABLE` to 10.

In conclusion, this trigger prevents users from inserting more than 10 records into `SAMPLE_TABLE`. It's a great example of how triggers can enforce business rules at the database level.

Trigger Example 2

```
CREATE TRIGGER CREDITS_EARNED
AFTER UPDATE OF TAKES ON (GRADE)
REFERENCING NEW ROW AS NROW
            OLD ROW AS OROW
FOR EACH ROW
WHEN NROW.GRADE <> 'F' AND NROW.GRADE IS NOT NULL
AND (OROW.GRADE = 'F' OR OROW.GRADE IS NULL)
BEGIN ATOMIC
    UPDATE STUDENT
    SET TOT_CRED= TOT_CRED +
    (
        SELECT CREDITS
        FROM COURSE
        WHERE COURSE.COURSE_ID= NROW.COURSE_ID
    )
    WHERE STUDENT.ID = NROW.ID;
END;
```

This code is creating a trigger named `CREDITS_EARNED` in Oracle Database. Here's what each part does:

1. `CREATE TRIGGER CREDITS_EARNED`: This line creates a trigger named `CREDITS_EARNED`.
2. `AFTER UPDATE OF TAKES ON (GRADE)`: This specifies when the trigger should be executed. In this case, it's set to run after an `UPDATE` operation on the `GRADE` column of the `TAKES` table.
3. `REFERENCING NEW ROW AS NROW OLD ROW AS OROW`: This line is creating aliases for the new and old versions of the row that's being updated. `NROW` refers to the new state of the row after the update, and `OROW` refers to the old state of the row before the update.
4. `FOR EACH ROW`: This means the trigger will execute for each row being updated.
5. `WHEN NROW.GRADE <> 'F' AND NROW.GRADE IS NOT NULL AND (OROW.GRADE = 'F' OR OROW.GRADE IS NULL)`: This is the condition for the trigger to fire. The trigger will only fire if the new grade is not 'F' and not null, and the old grade is either 'F' or null.
6. `BEGIN ATOMIC ... END;`: These keywords enclose the main logic of the trigger. The `ATOMIC` keyword ensures that the entire block of code is treated as a single transaction.
7. `UPDATE STUDENT SET TOT_CRED= TOT_CRED + (SELECT CREDITS FROM COURSE WHERE COURSE.COURSE_ID= NROW.COURSE_ID) WHERE STUDENT.ID = NROW.ID;`: This is the action that the trigger performs. It increases the `TOT_CRED` (total credits) of the student by the number of credits of the course with the same `COURSE_ID` as the new row. This only happens for the student with the same `ID` as the new row.

In conclusion, this trigger automatically updates the total credits of a student when their grade is updated to a passing grade. It's a great example of how triggers can automate database operations based on changes in data.

Statement Level Triggers

Statement-level triggers are one type of trigger that Oracle Database supports. As the name suggests, these triggers are fired once for each transaction, regardless of how many rows are affected by that transaction. This is in contrast to row-level triggers, which are fired once for each row affected by a transaction.

For example, if an `UPDATE` statement modifies 50 rows in a table, a statement-level trigger on that table would fire once. In contrast, a row-level trigger would fire 50 times.

Pros and Cons of Statement-Level Triggers

Pros:

1. **Efficiency:** Statement-level triggers can be more efficient than row-level triggers when a transaction affects many rows. This is because the trigger's code is only executed once, regardless of the number of rows affected.
2. **Simplicity:** Statement-level triggers can be simpler to write and understand than row-level triggers, especially when the trigger's action does not depend on the data in the individual rows affected by the transaction.

Cons:

1. **Lack of granularity:** Statement-level triggers do not have access to the data in the individual rows affected by the transaction. This means they cannot perform actions that depend on this data.
2. **Potential for unintended side effects:** Because statement-level triggers are fired once per transaction, they can potentially cause unintended side effects if not carefully written. For example, a statement-level trigger that inserts a row into another table every time it fires could result in unexpected duplicate rows if a transaction affects multiple rows.

`FOR EACH ROW` vs `FOR EACH STATEMENT`

In Oracle Database, the `FOR EACH ROW` clause is used to create a row-level trigger. This type of trigger fires once for each row affected by the triggering statement. For example, if an `UPDATE` statement modifies 50 rows in a table, a `FOR EACH ROW` trigger on that table would fire 50 times.

On the other hand, the `FOR EACH STATEMENT` clause is used to create a statement-level trigger. This type of trigger fires once for each triggering statement, regardless of the number of rows affected. For example, if an `UPDATE` statement modifies 50 rows in a table, a `FOR EACH STATEMENT` trigger on that table would fire once.

In conclusion, the choice between `FOR EACH ROW` and `FOR EACH STATEMENT` depends on the specific requirements of your application. If you need to perform an action for each row affected by a transaction, use a row-level trigger. If you need to perform an action once per transaction, regardless of the number of rows affected, use a statement-level trigger.

Statement Level Trigger Example

```
CREATE TRIGGER UPDATE_AVERAGE_GRADE
AFTER UPDATE ON GRADES
REFERENCING OLD TABLE AS OLD_GRADES NEW TABLE AS NEW_GRADES
FOR EACH STATEMENT
BEGIN ATOMIC
    DECLARE
        v_old_avg NUMBER;
        v_new_avg NUMBER;
    BEGIN
        SELECT AVG(GRADE) INTO v_old_avg FROM OLD_GRADES;
        SELECT AVG(GRADE) INTO v_new_avg FROM NEW_GRADES;
        IF v_new_avg > v_old_avg THEN
            INSERT INTO GRADE_CHANGES (CHANGE_DATE, CHANGE_TYPE, CHANGE_VALUE)
            VALUES (SYSDATE, 'INCREASE', v_new_avg - v_old_avg);
        ELSIF v_new_avg < v_old_avg THEN
            INSERT INTO GRADE_CHANGES (CHANGE_DATE, CHANGE_TYPE, CHANGE_VALUE)
            VALUES (SYSDATE, 'DECREASE', v_old_avg - v_new_avg);
        END IF;
    END;
END;
```

This code is creating a trigger named `UPDATE_AVERAGE_GRADE` in Oracle Database. Here's what each part does:

1. `CREATE TRIGGER UPDATE_AVERAGE_GRADE`: This line creates a trigger named `UPDATE_AVERAGE_GRADE`.
2. `AFTER UPDATE ON GRADES`: This specifies when the trigger should be executed. In this case, it's set to run after an `UPDATE` operation on the `GRADES` table.
3. `REFERENCING OLD TABLE AS OLD_GRADES NEW TABLE AS NEW_GRADES`: This line creates aliases for the old and new versions of the table that's being updated. `OLD_GRADES` refers to the state of the table before the update, and `NEW_GRADES` refers to the state of the table after the update.
4. `FOR EACH STATEMENT`: This means the trigger will execute once for each `UPDATE` statement, regardless of how many rows are updated.
5. `BEGIN ATOMIC ... END;`: These keywords enclose the main logic of the trigger. The `ATOMIC` keyword ensures that the entire block of code is treated as a single transaction.
6. `DECLARE v_old_avg NUMBER; v_new_avg NUMBER;`: These lines declare two variables `v_old_avg` and `v_new_avg` of type `NUMBER`.

7. `SELECT AVG(GRADE) INTO v_old_avg FROM OLD_GRADES; SELECT AVG(GRADE) INTO v_new_avg FROM NEW_GRADES;`: These lines calculate the average grade before and after the update and store the results into `v_old_avg` and `v_new_avg`, respectively.
8. `IF v_new_avg > v_old_avg THEN ... ELSIF v_new_avg < v_old_avg THEN ... END IF;`: This is a conditional statement. If the average grade has increased, it inserts a record into the `GRADE_CHANGES` table with the change type 'INCREASE' and the amount of the increase. If the average grade has decreased, it inserts a record with the change type 'DECREASE' and the amount of the decrease.

In conclusion, this trigger automatically tracks changes in the average grade and records increases and decreases in the `GRADE_CHANGES` table. It's a great example of how triggers can automate database operations based on changes in data.

Use of Triggers

- Using database triggers we can enforce business rules that can't be defined by using integrity constants.
- Using triggers we can gain strong control over the security.
- We can also collect statistical information on the table access.
- We can automatically generate values for derived columns such as auto-increment numeric primary key.
- Using database triggers we can prevent invalid transactions.

When Not to Use Triggers

Triggers, while powerful, are not always the best solution for every scenario. Here are some instances where using triggers might not be the best approach:

1. Maintaining Summary Data

In the past, triggers were often used to maintain summary data, such as the total salary of each department. However, modern databases provide built-in materialized view facilities to maintain summary data. These facilities are typically more efficient and easier to use than triggers.

2. Replicating Databases

Triggers were also used to replicate databases by recording changes to special relations (called change or delta relations) and having a separate process that applies the changes over to a replica. Nowadays, databases provide built-in support for replication, which is generally more reliable and efficient than using triggers.

3. Encapsulation Facilities

Encapsulation facilities can often be used instead of triggers. For example, you can define methods to update fields and carry out actions as part of the update methods instead of through a trigger. This approach can make your code easier to understand and maintain.

Risks of Using Triggers

There are also several risks associated with using triggers:

Unintended Execution

Triggers can execute unintentionally in certain scenarios, such as loading data from a backup copy or replicating updates at a remote site. However, trigger execution can be disabled before such actions to prevent this issue.

Transaction Failure

If a trigger contains an error, it can lead to the failure of critical transactions that set off the trigger. This can have serious consequences for your application.

Cascading Execution

Triggers can also lead to cascading execution, where one trigger fires another trigger, and so on. This can make your application's behavior difficult to predict and debug.

In conclusion, while triggers can be useful in many scenarios, they are not always the best tool for the job. It's important to understand the alternatives and risks associated with triggers before deciding to use them in your application.