

Constraints and Trigger

Constraint is a relationship among data elements enforced by the DBMS.

- Key constraints.

Triggers are operations that are executed when a specified condition occurs

- Ex. after insertion of a tuple.
- Easier to implement than complex constraints.
- Can think of as event-condition-action rules
 - A trigger is awakened when some event occurs
 - Once awakened, a condition is tested.
 - If the condition is satisfied, the action is carried out.

Kinds of Constraints

- Keys
- Foreign key, or referential-integrity constraint.
- Value-based constraints,

Constrain values of a particular attribute.

Tuple-based constraints.

Relationship among components.

Assertions: any SQL Boolean expression.

Single-Attribute Keys

Single attribute key refers to a unique identifier for a record within a table that consists of only one attribute or column.

Example:

```
CREATE TABLE Beers (name CHAR(20) PRIMARY KEY, Type CHAR(20));
```

Multiattribute Key

- A multiattribute key refers to a unique identifier for a record within a table that consists of more than one attribute or column.
- Unlike a single attribute key, which relies on only one attribute to uniquely identify a record, a multiattribute key combines two or more attributes to ensure uniqueness.

Example:

```
CREATE TABLE Employee (Name CHAR(20) Emp_nr VARCHAR(20), salary REAL,
PRIMARY KEY (Name, Emp_nr));
```

Foreign Keys

- A foreign key is a field or a combination of fields in one table that uniquely identifies a row of another table or the same table
- Sometimes values appearing in attributes of one relation must appear in certain attributes of another relation.
 - An attribute or set of attributes is a foreign key if it references some attribute(s) of a second relation.
 - This represents a constraint between relations

Expressing Foreign Keys

Definition in Table Creation:

- When creating a table in a database, foreign key constraint can be defined alongside the column definitions.
- It can be defined using the FOREIGN KEY keyword followed by the column name and the REFERENCES keyword indicating the referenced table and column.

```
CREATE TABLE Orders (OrderID INT PRIMARY KEY, CustomerID INT, OrderDate  
DATE, FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID));
```

Constraints

Foreign key constraints are defined when creating or altering tables. When creating a table, you specify foreign key constraints using the FOREIGN KEY constraint, referencing the primary key of another table.

Enforcing foreign key constraints is crucial in maintaining data integrity within a relational database.

Foreign key constraints ensure that relationships between tables are enforced, preventing actions that would violate those relationships, such as inserting records that reference non-existent primary key values.

Actions

When a foreign key constraint is violated, the DBMS enforces actions specified during constraint creation. Common actions include:

CASCADE: Automatically deletes or updates related rows in the child table when the corresponding rows in the parent table are deleted or updated.

RESTRICT: Prevents the deletion or update of a row in the parent table if it has related rows in the child table.

SET NULL: Sets the foreign key column in the child table to NULL when the corresponding row in the parent table is deleted or updated.

NO ACTION: Similar to RESTRICT, it prevents the deletion or update of a row in the parent table if it has related rows in the child table, without any additional action.

CASCADE

Automatically deletes or updates related rows in the child table when corresponding rows in the parent table are deleted or updated.

```
CREATE TABLE parent (parent_id INT PRIMARY KEY);
```

```
CREATE TABLE child (child_id INT PRIMARY KEY, parent_id INT, FOREIGN KEY  
(parent_id) REFERENCES parent(parent_id) ON DELETE CASCADE );
```

```
DELETE FROM parent WHERE parent_id = 1;
```

*Note: In this example, when we delete a row from the parent table, the corresponding rows in the child table are automatically deleted due to the **CASCADE** action.*

RESTRICT

Prevents the deletion or update of a row in the parent table if it has related rows in the child table.

```
CREATE TABLE parent (parent_id INT PRIMARY KEY);
```

```
CREATE TABLE child (child_id INT PRIMARY KEY, parent_id INT,  
FOREIGN KEY (parent_id) REFERENCES parent(parent_id) ON DELETE RESTRICT);
```

```
DELETE FROM parent WHERE parent_id = 1;
```

*Note: In this example, if we attempt to delete the row from the parent table that has related rows in the child table, the deletion will be prevented due to the **RESTRICT** action.*

SET NULL

Sets the foreign key column in the child table to NULL when the corresponding row in the parent table is deleted or updated.

```
CREATE TABLE parent ( parent_id INT PRIMARY KEY);
```

```
CREATE TABLE child (child_id INT PRIMARY KEY, parent_id INT, FOREIGN KEY
(parent_id) REFERENCES parent(parent_id) ON DELETE SET NULL );

DELETE FROM parent WHERE parent_id = 1;

SELECT * FROM child;
```

Note: In this example, when we delete a row from the parent table, the corresponding parent_id column in the child table is set to NULL due to the SET NULL action.

NO ACTION

Similar to RESTRICT, it prevents the deletion or update of a row in the parent table if it has related rows in the child table, without any additional action.

```
CREATE TABLE parent ( parent_id INT PRIMARY KEY);

CREATE TABLE child (child_id INT PRIMARY KEY, parent_id INT, FOREIGN KEY
(parent_id) REFERENCES parent(parent_id) ON DELETE NO ACTION);

DELETE FROM parent WHERE parent_id = 1;
```

Note: In this example, if we attempt to delete the row from the parent table that has related rows in the child table, the deletion will be prevented due to the NO ACTION action.

Monitoring and Reporting

Regular monitoring and reporting on foreign key constraints can help identify potential issues with data integrity and performance.

Database administrators should regularly check for constraint violations and optimize where necessary.

Check Constraint

Check constraint is a type of constraint that defines a condition that must be true for every row in a table.

Check constraints are used to enforce data integrity rules or conditions on individual columns within a table.

```
CREATE TABLE employees (emp_id INT PRIMARY KEY, emp_name  
VARCHAR(50),
```

```
age INT, CONSTRAINT check_age CHECK (age BETWEEN 18 AND 65));
```

In the above example, the CHECK constraint check_age ensures that the age column in the employees table contains values that fall within the specified range (between 18 and 65).

If an attempt is made to insert or update a row with an age value outside this range, the DBMS will raise an error, and the operation will be rejected.

Check constraints provide a powerful mechanism for enforcing data integrity rules at the column level, ensuring that only valid data is stored in the database.

They can be used to enforce various types of conditions, such as range checks, value comparisons, or pattern matching.

Domain Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
- Domain constraints are the most elementary form of integrity constraint.
- They test values inserted in the database, and test queries to ensure that the comparisons make sense.
- New domains can be created from existing data types

E.g. **create domain** *Dollars* **numeric**(12, 2)

create domain *Pounds* **numeric**(12,2)

Assertion

- Assertion is a logical expression that defines a condition that must be true for the database to be in a consistent state.
- Assertions are used to enforce integrity constraints that cannot be expressed using simple column-based constraints like primary key, foreign key, or check constraints.

Example:

```
CREATE TABLE departments (dept_id INT PRIMARY KEY, dept_name VARCHAR(50)
);
```

```
CREATE TABLE employees (emp_id INT PRIMARY KEY, emp_name VARCHAR(50),
dept_id INT, CONSTRAINT fk_dept FOREIGN KEY (dept_id) REFERENCES
departments(dept_id) );
```

```
CREATE ASSERTION department_existence CHECK (NOT EXISTS (SELECT * FROM
employees WHERE dept_id NOT IN (SELECT dept_id FROM departments)));
```

- In the above example, the department_existence assertion ensures that every dept_id referenced in the employees table must exist in the departments table.
- If there's any dept_id in the employees table that doesn't exist in the departments table, the assertion will be violated, and the database won't allow the operation that caused the violation to proceed.
- Assertions provide a flexible way to enforce complex integrity constraints that involve relationships between multiple tables.

Constraints Vs Assertion - Purpose

Constraints

- Constraints are used to enforce rules or conditions on individual columns or combinations of columns within a table.
- They ensure that data inserted or updated in the database meets certain criteria, such as primary key uniqueness, foreign key relationships, or data range limitations.
- Constraints operate at the column or table level.
- They are applied directly to columns or tables and enforce rules specific to those columns or tables.

Assertions

- Assertions are used to enforce more complex integrity rules that cannot be expressed using simple column-based constraints.
- They typically involve conditions that span multiple tables or involve complex logical expressions.
- Assertions operate at the database level.
- They are defined separately from individual tables and can involve conditions that span multiple tables or involve data from different parts of the database schema.

Triggers

- Triggers are event driven program
- Not necessary to execute manually
- Triggers are automatically executed when the event occurs

- There are 12 events are there in PL/SQL
 - Before insert
 - Before delete
 - Before update
 - After insert
 - After delete
 - After update

Syntax for Trigger

CREATE [OR REPLACE] TRIGGER trigger_name

{BEFORE | AFTER | INSTEAD OF } {INSERT [OR] | UPDATE [OR] | DELETE}

[OF col_name]

ON table_name

[REFERENCING OLD AS o NEW AS n] [FOR EACH ROW]

WHEN (condition)

DECLARE

<Declaration-statements>

BEGIN

<Executable-statements>

EXCEPTION

<Exception-handling-statements>

END;

CREATE [OR REPLACE] TRIGGER trigger_name – Creates or replaces an existing trigger with the trigger_name.

{BEFORE | AFTER | INSTEAD OF} – This specifies when the trigger will be executed.

The INSTEAD OF clause is used for creating trigger on a view.

{INSERT [OR] | UPDATE [OR] | DELETE} – This specifies the DML operation.

[OF col_name] – This specifies the column name that will be updated.

[ON table_name] – This specifies the name of the table associated with the trigger.

[REFERENCING OLD AS o NEW AS n] – This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.

[FOR EACH ROW] – This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.

WHEN (condition) – This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

Example : Display the salary difference in emp table

```
create or replace trigger display_salary_changes
```

```
before delete or insert or update on emp
```

```
for each row
```

```
when (new.empno > 0)
```

```
declare
```

```
    sal_diff number;
```

```
begin
```

```
    sal_diff := :new.sal - :old.sal;
```

```
    dbms_output.put_line('old salary: ' || :old.sal);
```

```
    dbms_output.put_line('new salary: ' || :new.sal);
```

```
    dbms_output.put_line('salary difference: ' || sal_diff);
```

end;

Output

SQL> update emp set sal = 1111 where empno = 7900;

Old salary: 950

New salary: 1111

Salary difference: 161

1 row updated.

SQL> insert into emp values (1234,'NANTHA','MANAGER',7839,'23-MAR-83',4000,NULL,30);

Old salary:

New salary: 4000

Salary difference:

1 row created.

SQL> delete from emp where empno=1234;

1 row deleted.

Trigger

Trigger Example on Insert

-- Create employees table

```
CREATE TABLE employees (  
    emp_id INT PRIMARY KEY AUTO_INCREMENT,  
    emp_name VARCHAR(50),  
    emp_salary DECIMAL(10, 2)  
);
```

-- Create audit_log table to store insert events

```

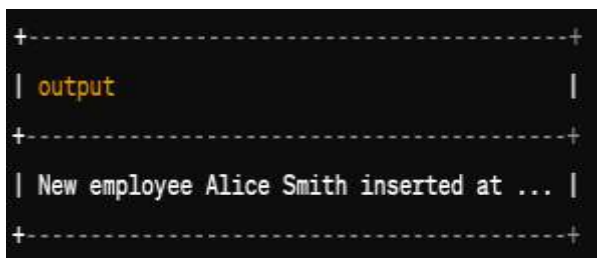
CREATE TABLE audit_log (
    log_id INT PRIMARY KEY AUTO_INCREMENT,
    action_type VARCHAR(10),
    action_time TIMESTAMP
);

CREATE TRIGGER log_insert_employee
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
    INSERT INTO audit_log (action_type, action_time)
    VALUES ('INSERT', NOW());

    SELECT CONCAT('New employee ', NEW.emp_name, ' inserted at ', NOW()) AS
output;
END;

INSERT INTO employees (emp_name, emp_salary) VALUES ('Alice Smith', 60000.00);

```



```

+-----+
| output |
+-----+
| New employee Alice Smith inserted at ... |
+-----+

```

Trigger Example on Delete

In above example:

- The employees table stores information about employees, and the audit_log table is used to log actions.

- We create a trigger named log_delete_employee that activates before a DELETE operation on the employees table.
- For each row deleted from the employees table, the trigger inserts a record into the audit_log table, indicating that a DELETE action has occurred along with the current timestamp.
- The trigger also provides output indicating which employee was deleted and the timestamp of the deletion.

Trigger Example on Update

```
CREATE TRIGGER log_update_employee
BEFORE UPDATE ON employees
FOR EACH ROW
BEGIN
    INSERT INTO audit_log (action_type, action_time)
    VALUES ('UPDATE', NOW());
    SELECT CONCAT('Employee ', OLD.emp_name, ' updated at ', NOW()) AS
output;
END;
```

In above example:

- The employees table stores information about employees, and the audit_log table is used to log actions.
- We create a trigger named log_update_employee that activates before an UPDATE operation on the employees table.

- For each row updated in the employees table, the trigger inserts a record into the audit_log table, indicating that an UPDATE action has occurred along with the current timestamp.
- The trigger also provides output indicating which employee was updated and the timestamp of the update.

```
+-----+
| output |
+-----+
| Employee Alice Smith updated at ... |
+-----+
```

Constraints and Triggers

Trigger Example

-- Create the orders table

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    total_amount DECIMAL(10, 2)
);
```

-- Create the order_history table to store deleted orders

```
CREATE TABLE order_history (
    order_id INT PRIMARY KEY,
    customer_id INT,
    total_amount DECIMAL(10, 2),
    deleted_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```
CREATE TRIGGER archive_deleted_orders  
  
AFTER DELETE ON orders  
  
FOR EACH ROW  
  
BEGIN  
  
    INSERT INTO order_history (order_id, customer_id, total_amount)  
  
    VALUES (OLD.order_id, OLD.customer_id, OLD.total_amount);  
  
END;
```

In above example:

- The orders table stores information about active orders, and the order_history table is used to archive deleted orders.
- We create a trigger named archive_deleted_orders that activates after a DELETE operation on the orders table.
- For each row deleted from the orders table, the trigger inserts a record into the order_history table, archiving the deleted order along with the deletion timestamp.

Uses of triggers in this example

Data Archival: Triggers are useful for archiving or logging data changes. In this case, the trigger archives deleted orders into the order_history table, maintaining a historical record of deleted orders.

Data Integrity: Triggers can enforce data integrity constraints. For example, you can use triggers to prevent the deletion of critical data or to enforce referential integrity between tables.

Audit Logging: Triggers can be used for auditing purposes. You can create triggers to log changes made to specific tables, allowing you to track who made changes and when.

Derived Data Maintenance: Triggers can automatically update derived data based on changes to other data. For example, you can use triggers to update summary tables or calculate aggregates when underlying data changes.

Trigger Example

```
CREATE TABLE students (  
    student_id INT PRIMARY KEY AUTO_INCREMENT,  
    student_name VARCHAR(50),  
    date_of_birth DATE,  
    grade VARCHAR(10)  
);  
  
CREATE TABLE student_log (  
    log_id INT PRIMARY KEY AUTO_INCREMENT,  
    student_id INT,  
    old_name VARCHAR(50),  
    new_name VARCHAR(50),  
    old_date_of_birth DATE,  
    new_date_of_birth DATE,  
    old_grade VARCHAR(10),  
    new_grade VARCHAR(10),  
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP);  
  
CREATE TRIGGER log_student_update
```



```

AFTER UPDATE ON students
FOR EACH ROW
BEGIN
    IF OLD.student_name != NEW.student_name OR OLD.date_of_birth !=
NEW.date_of_birth OR OLD.grade != NEW.grade THEN
        INSERT INTO student_log (student_id, old_name, new_name,
old_date_of_birth, new_date_of_birth, old_grade, new_grade)
        VALUES (OLD.student_id, OLD.student_name, NEW.student_name,
OLD.date_of_birth, NEW.date_of_birth, OLD.grade, NEW.grade);
    END IF;
END;

```

In above example:

- The students table stores information about students, and the student_log table is used to log updated student information.
- We create a trigger named log_student_update that activates after an UPDATE operation on the students table.
- For each row updated in the students table, the trigger checks if any of the student's information (name, date of birth, or grade) has changed. If there are changes, it inserts a record into the student_log table, logging the old and new values along with the update timestamp.

Trigger Example-1

```

CREATE TRIGGER enforce_salary_threshold
BEFORE UPDATE ON employees
FOR EACH ROW

```

```
BEGIN
```

```
IF NEW.emp_salary < 50000.00 THEN
```

```
    SET NEW.emp_salary = 50000.00; -- Set salary to threshold value
```

```
END IF;
```

```
END;
```

In above example:

- We have a table called employees with columns for employee ID, name, and salary.
- We insert some sample data into the employees table.
- We create a trigger named enforce_salary_threshold that activates before an UPDATE operation on the employees table.
- For each row being updated, the trigger checks if the new salary value (NEW.emp_salary) is less than the threshold value of \$50,000. If it is, the trigger sets the new salary value to the threshold value of \$50,000.

```
UPDATE employees SET emp_salary = 45000.00 WHERE emp_id = 1;
```

- Upon executing the above UPDATE statement, the trigger will activate, and the employee's salary will be automatically adjusted to the threshold value of \$50,000.
- This example demonstrates how triggers can be used to enforce business rules, such as maintaining a minimum salary threshold for employees, and ensure data integrity within the database.

Advantage of Triggers

1. Database object rules are established by triggers, which cause changes to be undone if they are not met.

2. The trigger will examine the data and, if necessary, make changes.
3. We can enforce data integrity thanks to triggers.
4. Data is validated using triggers before being inserted or updated.
5. Triggers assist us in maintaining a records log.
6. Due to the fact that they do not need to be compiled each time they are run, triggers improve the performance of SQL queries.
7. The client-side code is reduced by triggers, saving time and labor.
8. Trigger maintenance is simple.

Disadvantage of Triggers

1. Only triggers permit the use of extended validations.
2. Automatic triggers are used, and the user is unaware of when they are being executed. Consequently, it is difficult to troubleshoot issues that arise in the database layer.
3. The database server's overhead may increase as a result of triggers.
4. In a single CREATE TRIGGER statement, we can specify the same trigger action for multiple user actions, such as INSERT and UPDATE.
5. Only the current database is available for creating triggers, but they can still make references to objects outside the database.