Integrity Constraints in SQL

Relational Database Schema

- A *database schema* is a set of relation schemas and a set of ***integrity constraints***

- Integrity Constraints
  - *Structural* Integrity Constraints
    - key constraints: uniqueness of keys
    - entity integrity constraint: no primary key value can be **NULL**
    - referential integrity constraint
  - *Semantic* Integrity Constraints
    - E.g., ??

Structural Constraints in SQL

- Constraints (on Attributes):
  - NOT NULL
  - DEFAULT value
    - without the DEFAULT-clause, the default value is NULL
  - PRIMARY KEY (attribute-list)
  - UNIQUE (attribute list)
    - allows the specification of alternative key
  - FOREIGN KEY (key) REFERENCES table (key)

Referential Triggered Actions

- Actions if a Referential Integrity constraint is violated
  - SET NULL
  - CASCADE (propagate action)
  - SET DEFAULT
- Qualify actions by the triggering condition:
  - ON DELETE
  - ON UPDATE
- Note: Oracle does not support ON UPDATE & SET DEFAULT

Create Table with RI Trigger Actions

```
CREATE TABLE  LIBRARIAN                /* or Micro_db.LIBRARIAN */
(    Name     name_dom,
     SSN      ssn_dom,
     Section  INTEGER,
     Address  address_dom,
     Gender   gender_dom,
     Birthday DATE,
     Salary   DEC(8,2),
     CONSTRAINT librarian_PK  PRIMARY KEY (SSN),
     CONSTRAINT librarian_FK
       FOREIGN KEY (Section)  REFERENCES SECTION (SNO)
       On Delete SET DEFAULT On Update CASCADE
);
```


Semantic Integrity Constraints

- A constraint is expressed as a Predicate, a condition similar to the one at the WHERE-clause of a query
- Three DDL constructs
  - Checks
  - Assertions
  - Triggers

Check Constraints

- CHECK *prohibits* an operation on a table that would violate the constraint. It is a *local* constraint.

```
CREATE TABLE SECTION
 ( SectNo sectno_dom,
   Name section_dom,
   HeadSSN ssn_dom,
   Budget budget_dom,
  CONSTRAINT section_PK
     PRIMARY KEY (SectNo),
  CONSTRAINT section_FK
    FOREIGN KEY (HeadSSN) REFERENCES LIBRARIAN(SSN))
```


Check Constraints…

```
CONSTRAINT section_budget_IC1
   CHECK ((Budget >= 0) AND (Budget IS NOT NULL)),

CONSTRAINT section_budget_IC2
   CHECK (NOT EXISTS
          (SELECT * FROM SECTION WHERE budget <
          (SELECT SUM (Salary) FROM LIBRARIAN))),

 CONSTRAINT Head_Lib_IC3
    CHECK (HeadSSN <> ALL (SELECT SSN FROM Retiree))
);
```

Assertions

- Similar to CHECK but they are **global** constraints

  CREATE OR REPLACE ASSERTION <assertion_name>
  CHECK <Predicate> [Mode of Evaluation];

  - **Predicate** usually involves EXISTS and NOT EXISTS

- E.g., **CREATE OR REPLACE ASSERTION** budget_constraint

  CHECK (**NOT EXISTS**

  (SELECT * FROM SECTION WHERE budget <
  (SELECT SUM (Salary) FROM LIBRARIAN)));

  VQuery that violates IC

- Dropping an assertion…

  DROP ASSERTION budget_constraint;

Triggers

- A trigger consists of 3 parts:
  1. Event(s),
  2. Condition, and
  3. Action

- E.g., Notify the Dean whenever the number of students in any major exceeds 1800

- Triggers could be associated with a table or a view

Triggers vs. Assertions

- Assertion
  - Condition must be true for each database state
  - DBMS rejects operations that violate such condition

- Trigger
  - DBMS takes a certain **action** when condition is true
  - Action could be: stored procedure, SQL statements, Rollback, etc.

My First Trigger

- Notify the Dean when the # of students in any major exceeds 1800

CREATE TRIGGER Major_Limit

Event(s)

Condition

Action

Example

- Notify the Dean when the # of students in any major exceeds 1800

CREATE TRIGGER Major_Limit

Event(s)

```
WHEN( EXISTS (
                SELECT Major_Code, COUNT(*)
                FROM Student
                GROUP BY Major_Code
                HAVING COUNT(*) > 1800 ))
```

Action

Example

- Notify the Dean when the # of students in any major exceeds 1800

CREATE TRIGGER Major_Limit

Event(s)

```
WHEN( EXISTS (
                SELECT Major_Code, COUNT(*)
                FROM Student
                GROUP BY Major_Code
                HAVING COUNT(*) > 1800 ))
```

```
CALL email_dean(Major_code);
```


Example

- Notify the Dean when the # of students in any major exceeds 1800

CREATE TRIGGER Major_Limit

```
AFTER INSERT OR UPDATE OF Major_Code
ON Student
```

```
WHEN( EXISTS (
                SELECT Major_Code, COUNT(*)
                FROM Student
                GROUP BY Major_Code
                HAVING COUNT(*) > 1800 ))
```

```
CALL email_dean(Major_code);
```


Example: Assertions Vs. Triggers

- **CREATE OR REPLACE ASSERTION** budget_constraint
  CHECK (**NOT EXISTS**
  (SELECT * FROM SECTION WHERE budget <
  (SELECT SUM (Salary) FROM LIBRARIAN)));

- **CREATE OR REPLACE TRIGGER** budget_constraint_trigger
  after INSERT, UPDATE of Salary
  ON LIBRARIAN
  WHEN (**EXISTS** (SELECT * FROM SECTION WHERE budget <
  (SELECT SUM (Salary) FROM LIBRARIAN))
  ROLLBACK;

Triggers (SQL99)

- CREATE or REPLACE TRIGGER <trigger-name>
  <time events>
  ON <list-of-tables>
  [REFERENCING { NEW | OLD } AS <user-name>]
  [FOR EACH { ROW | STATEMENT }]
  [WHEN (<Predicate>)]
  <action>
- time: **before** or **after**
- events: **Insert, Delete, Update [of <list of attributes>]**
- **NEW & OLD** refer to new & old (existing) tuples/table respectively
- The REFERENCING clause assigns aliases to NEW and OLD
- action: Stored procedure or
  BEGIN ATOMIC {<SQL procedural statements>} END

Creating triggers in Postgres

- CREATE TRIGGER *trig_name*
  time event(s)
  ON { *table_name* | view_name }
  [REFERENCING { NEW | OLD } TABLE <user-name>]
  [FOR EACH { ROW | STATEMENT }]
  [WHEN (*condition*)]
  EXECUTE {FUNCTION | PROCEDURE} *func_name* ();

Row-level triggers in PostgreSQL

- **Example of Row-level Trigger**

```
CREATE TRIGGER Name_Trim
   BEFORE INSERT
   ON Student
   FOR EACH ROW
     WHEN (NEW.Name IS NOT NULL)
   EXECUTE FUNCTION trim_spaces_name();
```

- **OLD TABLE** only for **UPDATE** or **DELETE**
- **NEW TABLE** only for **UPDATE** or **INSERT**
- No **REFERENCING** and no **UPDATE** of specific columns

Statement-level triggers in PostgreSQL

- **Example of Table-level Trigger**

```
CREATE TRIGGER trigger_audit
   AFTER INSERT OR DELETE OR UPDATE
   ON Student
   EXECUTE FUNCTION update_log();
```

- **REFERENCING** is valid only for statement-level triggers
  - only for **one** event and
  - only for **AFTER** event time

When triggers can fire in Postgres

- *time*
  - BEFORE
  - AFTER
  - INSTEAD OF
- *event*
  - INSERT
  - DELETE
  - UPDATE [OF *att_name* [, …]]
  - TRUNCATE
- NEW and OLD are valid references in the trigger function

Compatibility

TIME	EVENT	ROW	STATEMENT
BEFORE	INSERT, UPDATE, DELETE	Tables	Tables and views
	TRUNCATE	—	Tables
AFTER	INSERT, UPDATE, DELETE	Tables	Tables and views
	TRUNCATE	—	Tables
INSTEAD OF	INSERT, UPDATE, DELETE	Views	—
	TRUNCATE	—	—

Enable & Disable Triggers in PostgreSQL

- Enable/Disable All Triggers:

```
ALTER TABLE <table_name> ENABLE TRIGGER ALL;
ALTER TABLE <table_name> DISABLE TRIGGER ALL;
```

  - E.g., ALTER TABLE Librarian DISABLE TRIGGER ALL;

- Enable/Disable Individual Trigger

```
ALTER TABLE <table_name>
 ENABLE | DISABLE TRIGGER <trigger_name>;
```

  - E.g., ALTER TABLE Librarian
    DISABLE TRIGGER librarian_salary_trigger;

Dropping triggers in PostgreSQL

- The DROP TRIGGER statement in PostgreSQL is incompatible with the SQL standard. In the SQL standard, trigger names are not local to tables.

```
DROP TRIGGER [ IF EXISTS ] trig_name
    ON table_name [ CASCADE | RESTRICT ];
```

  - **CASCADE:** Automatically drop objects that depend on the trigger.
  - **RESTRICT:** Refuse to drop the trigger if any objects depend on it. This is the default.

Mutating Trigger

- Recursive call of triggers is not permitted
- Table read in a trigger it cannot be updated

```
CREATE FUNCTION increment() RETURNS TRIGGER
 AS $$
 BEGIN
       SELECT MAX(ID) + 1 INTO NEW.ID
       FROM Students;
       RETURN NEW;
 END;
 $$ LANGUAGE plpgsql;

CREATE TRIGGER bad_auto_sid
       AFTER INSERT ON Students
       FOR EACH ROW
       EXECUTE FUNCTION increment();
```


Mutating Trigger

- Recursive call of triggers is not permitted
- Table read in a trigger it cannot be updated

```
CREATE FUNCTION increment() RETURNS TRIGGER
 AS $$
 BEGIN
       SELECT MAX(ID) + 1 INTO NEW.ID
       FROM Students;
       RETURN NEW;
 END;
 $$ LANGUAGE plpgsql;

CREATE TRIGGER good_auto_sid
       BEFORE INSERT ON Students
       FOR EACH ROW
       EXECUTE FUNCTION increment();
```

- INTO: the tuple assignment operator in PL/SQL

Final note on IC

- Assertions and Checks Vs. Triggers
  - Assertions and Checks support the declarative approach of supporting Integrity Constraints
  - Triggers combine the declarative and procedural approach of implementing integrity constraints