5. Integrity Constraints

Contents

Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.

  Example:
  If a supplier name occurs in the relation offers, then this supplier name must also occur in the relation SUPPLIERS.

- Formal definition:
  - Let $r(R)$ and $s(S)$ be relations with primary keys K_1 and K_2 respectively.
  - The subset α of attributes of S is a *foreign key* referencing K_1 in r, if for every tuple t in s there must be a tuple t' in r such that $t'[K_1] = t[\alpha]$.
  - Referential integrity constraint: $\pi_\alpha(S) \subseteq \pi_{K_1}(R)$

- Referential Integrity in the ER Model:
  - Consider a relationship set R between two entity sets E_1 and E_2. The relation schema corresponding to R includes the primary keys K_1 of E_1 and K_2 of E_2.
  - Then K_1 and K_2 form the foreign keys to the relation schemas for E_1 and E_2, respectively.

Specification of Referential Integrity

- Referential Integrity Constraints are specified as part of the SQL **create table** statement (or added through **alter table**).
- Example in PostgreSQL DDL:

```
create table CUSTOMERS (
    FName      varchar(20),
    LName      varchar(40),
    CAddress   varchar(80) not null,
    Account    real,
    constraint cust_pk primary key (FName, LName)
);

create table PRODUCTS (
    Prodname   varchar(80) constraint prod_pk primary key,
    Category   char(20)
);

create table SUPPLIERS (
    SName      varchar(60) constraint supp_pk primary key,
    SAddress   varchar(60) not null,
    Chain      varchar(30)
);
```

```
create table offers (
    Prodname    varchar(80) constraint ref_prod
                                    references PRODUCTS,
    SName       varchar(60) constraint ref_supp
                                    references SUPPLIERS,
    Price       real not null,
constraint pk_offers primary key(Prodname, SName)
);
```

```
create table orders (
    FName       varchar(20) not null,
    LName       varchar(40) not null,
    SName       varchar(60) not null references SUPPLIERS,
    Prodname    varchar(80) references PRODUCTS,
    Quantity    integer     check(Quantity > 0),
foreign key(FName, LName) references CUSTOMERS,
primary key(FName, LName, Prodname, SName)
);
```

Note that foreign key constraints above allow null values, that is, the constraint **not null** should be added.

- Tests must be executed after database modifications to preserve referential integrity (and other constraints):
  - Assume referential integrity constraint $\pi_\alpha(S) \subseteq \pi_K(R)$

    $\pi_{\text{Prodname}}(\texttt{offers}) \subseteq \pi_{\text{Prodname}}(\texttt{PRODUCTS})$ or
    $\pi_{\text{FName, LName}}(\texttt{orders}) \subseteq \pi_{\text{FName, LName}}(\texttt{CUSTOMERS})$

  - If a tuple t is **inserted** into S, the DBMS must verify whether there is a tuple t' in R such that $t'[K] = t[\alpha]$; that is, $t[\alpha] \in \pi_K(R)$. If not, the insertion is rejected.

  - If a tuple t' is **deleted** from R, the system must check whether there are tuples in S that reference t'. That is, $\sigma_{\alpha=t'[K]}(S)$ must be empty. If not, the deletion is rejected or the tuples that reference t' must themselves be deleted (if cascading deletions are possible).

    create table offers (
    Prodname **varchar**(80) **references** PRODUCTS
    on delete cascade,
    . . .);

  - There are two cases for **updates**:

    (1) Update of referencing attributes (i.e., on $\pi_\alpha(S)$).
    $\rightarrow$ treated in the same way as an insertion into S.

    (2) Updates on the referenced attributes (i.e., on $\pi_K(R)$).
    $\rightarrow$ treated similar to the delete case, may include
    on update cascade for referencing attributes.

Assertions

- An *assertion* is a predicate expressing a condition that we want the database always to satisfy.
- Assertions are included in the SQL standard. Syntax:
  create assertion <name> **check** (<predicate>)
- When an assertion is specified, the DBMS tests for its validity. This testing may introduce a significant amount of computing overhead (query evaluation), thus assertions should be used carefully
- Note that assertions are not offered in PostgreSQL (or, indeed, in most other systems). Still, the concept is useful to understand.
- Example:

  For each product, there must be at least two suppliers.

```
create assertion two_suppliers check
(not exists (select * from offers O1
             where not exists
                   (select * from offers O2
                    where O1.SName <> O2.SName
                      and O1.Prodname = O2.Prodname))
)
```

Units of Enforcing Integrity Constraints

- Question: When does the DBMS verify whether an integrity constraint is violated?
- Approach 1: After a single database modification (**insert, update** or **delete** statement)
  $\hat{=}$ *immediate mode*

  Approach 2: At the end of a transaction, i.e., after a sequence of database modifications (**begin transaction** <sequence of db modifications> **end transaction** (or **commit**))
  $\hat{=}$ *deferred mode*
- Certain combinations of integrity constraints can only be verified in deferred mode, i.e., constraint violating (intermediate) database states within a transaction are allowed.

Given an integrity constraint I, which database modifications can violate the integrity constraint?

$\leadsto$ the *critical operations* for an integrity constraint

Example: For each `PRODUCT`, there must be a `SUPPLIER` who `offers` the `PRODUCT`. Which operations, on which relations, can violate I?

5.2 Triggers

- A *trigger* is a statement (procedure) that is executed automatically by the DBMS whenever a specified event occurs.

- Triggers can be used for
  - Maintaining integrity constraints
  - Auditing of database actions (e.g., data modifications)
  - Propagation of database modifications

- To design a trigger, one has to specify
  - the event and condition under which the trigger is to be executed, and
  - the action(s) to be performed when the trigger executes

- Because of this structure, triggers are sometimes also called *Event-Condition-Action* (ECA) rules

- Triggers are included in the SQL:2003 standard and they are offered by almost all commercial database management systems (though often using a syntax somewhat different from the standard).

Triggers in the SQL:2003 standard

- Format:

  create trigger <name>
  {**before**|**after**} <trigger event(s)>
  on <table> [**referencing** <transition table or variable list>]
  [**for each** {**row** | **statement**}]
  [**when** <condition>]
  <triggered SQL statement>

  A trigger is *fired* if <trigger event(s)> occurred before/after an event in a transaction (immediate/deferred);
  A trigger is *executed* if <condition> evaluates to true.

- Using triggers for integrity maintenance:

  create trigger <name>
  after <critical database modification(s)>
  when <integrity constraint violated>
  <action(s)>

  with <action> – **rollback** of the violating transaction (passive), or
  – repairing operations (active)

- Important feature underlying triggers:

  The DBMS keeps track of modifications done by a transaction using so-called *transition tables*.

- Given a relation R. The idea is that the DBMS maintains four relations (transition tables) for R during the execution of a transaction T.
  - $R_{\text{deleted}} \hat{=}$ tuples deleted from R during T
  - $R_{\text{inserted}} \hat{=}$ tuples inserted into R during T
  - $R_{\text{updated\_old}} \hat{=}$ values of updated tuples before T
  - $R_{\text{updated\_new}} \hat{=}$ values of updated tuples after T

- The modified relation R' after transaction T thus can be obtained as

  $R' = R - R_{\text{deleted}} \cup R_{\text{inserted}} - R_{\text{updated\_old}} \cup R_{\text{updated\_new}}$.

- Verification of integrity constraints can be optimized if transition tables are provided. **Assumption:** Before the transaction, all integrity constraints are satisfied (i.e., there were no violations).

  Example:
  - Assume the constraint *Every product must be offered by at least one supplier*.
  - Critical operations are **insert** into `PRODUCTS`, **delete** from `offers` (and **update** on `offers` and `PRODUCTS`)
  - Only products inserted by the transaction need to be verified $\rightarrow$ tuples in $\texttt{PRODUCTS}_{\text{inserted}}$ (analogous for deletions from `offers`)

Example Trigger Definitions in SQL:2003 (not PostgreSQL!)

- Format:

```
create trigger <name>
{before|after} <trigger event(s)>
on <table> [referencing <transition table or variable list>]
[ for each {row | statement} ]
[ when <condition> ]
<triggered SQL statement>
```

with <trigger event(s)> one or more events from **insert**, **update** [**of** (<list of columns>)], or **delete**.

1. "The balance of a customer's account must not fall below -$10,000."

```
create trigger bad_account
after insert or update of(Account) on CUSTOMERS
referencing new table as ins_customer
when ( exists ( select * from ins_customer
                where Account < -10000) )
begin
    rollback;
end
```

`ins_customer` is a transition table that only stores the new values of tuples inserted into / updated in `CUSTOMERS` during the transaction.

2. "For each product, there must be an offer."

```
create trigger bad_product
after insert on PRODUCTS
referencing new table as new_prods
when ( exists ( select * from new_prods n where not exists
                ( select * from offers
                  where n.Prodname = Prodname)))
begin . . .
```

3. "The quantity of an order can only be increased."

```
create trigger no_decrease_quantity
after update of(Quantity) on orders
referencing new row as nrow,  old row as orow
for each row
when ( nrow.Quantity < orow.Quantity)
begin
    update orders O set Quantity = orow.Quantity
    where O.SName = nrow.SName
      and O.FName = nrow.FName
      and O.LName = nrow.LName
      and O.Prodname = nrow.Prodname
end
```

orow and nrow are transition variables that hold the value of the old and new updated tuple (requires **for each row** clause)

Triggers in PostgreSQL

- Triggers in PostgreSQL are based on user-defined functions in trigger body
- Components of a trigger:
  - *trigger name*
    create trigger <trigger name>
  - *trigger time point*
    before | **after**
  - *triggering event(s)* (can be combined via **or**)
    {**insert**|**update** [**of** <column(s)>]|**delete**} **on** <table>
  - *trigger type* (optional)
    for each row
  - *trigger restriction* (only for **for each row** triggers !)
    when (<condition>)
  - *trigger body*
    execute procedure <function_name> (<arguments>)

- There are two types of triggers:

– *statement level trigger:* trigger is fired (executed) before/after a statement (**update**, **insert**, **delete**)

– *row level trigger*: trigger is fired (executed) before/after each single modification (one tuple at a time)

Special features of a row level trigger:

- may include **when** clause containing a simple condition
- old and new values of tuple can be referenced using **old**.<column> and **new**.<column>

  event = **delete** → only **old**.<column>,
  event = **insert** → only **new**.<column>

  in PL/pgSQL block, e.g., **if old**.`SAL` < **new**.`SAL` **then** . . . or

  new.`SAL` := **new**.`SAL` ∗ 1.05

- There are 12 different basic trigger types in PostgreSQL:

event	trigger time point		trigger type	
	before	**after**	statement	row
insert	X	X	X	X
update	X	X	X	X
delete	X	X	X	X

Example Triggers in PostgreSQL, Using PL/pgSQL

In the following we assume the relations:

EMP(Empno, EName, Job → SALGRADE, Mgr, Hiredate
Sal, Deptno → DEPT)

DEPT(Deptno, Dname, Loc, Budget)

SALGRADE(Job, Minsal, Maxsal)

Let's see how to implement the following integrity constraint:

"The salary of an employee different from the president cannot be decreased and must also not be increased by more than 10%. Furthermore, depending on the job title, each salary must lie within a certain salary range."

This constraint might be affected by operations on EMP and SALGRADE, so we need *two* triggers. . .

(1) Trigger for operations on EMP

```
create function check_salary_EMP() returns trigger as '
declare
  minsal real;
  maxsal real;
begin
   -- retrieve minimum and maximum salary for job
   select S.minsal, S.maxsal into minsal, maxsal from SALGRADE S
   where S.job = new.job;
   -- If the new salary has been decreased or does not
   -- lie within the salary range, raise an exception
   if (new.sal < minsal or new.sal > maxsal) then
      raise exception ''Salary range exceeded'';
   elsif (TG_OP = ''UPDATE'') then
      if (new.sal < old.sal) then
         raise exception ''Salary has been decreased'';
      elsif (new.sal > 1.1 * old.sal) then
         raise exception ''More than 10 percent salary increase'';
      end if;
   end if;
   return new;
end;
' language plpgsql;

create trigger check_salary_EMP
after insert or update of Sal, Job on EMP
for each row
when ( upper(new.JOB) != 'PRESIDENT') -- trigger restriction
execute procedure check_salary_EMP();
```

(2) Trigger for operations on SALGRADE

```
create function check_salary_SALGRADE() returns trigger as '
declare
  job_emps  int;
begin
   -- Are there employees whose salary does not lie within
   -- the modified salary range?
   select count(*) into job_emps from EMP
   where JOB = new.JOB
     and SAL not between new.MINSAL and new.MAXSAL;
   if (job_emps != 0) then  -- restore old salary ranges
     new.MINSAL := old.MINSAL;
     new.MAXSAL := old.MAXSAL;
   end if;
   return new;
end;
' language plpgsql;

create trigger check_salary_SALGRADE
before update on SALGRADE
for each row
when (new.MINSAL > old.MINSAL or new.MAXSAL < old.MAXSAL)
  -- since only restricting a salary range
  -- can cause a constraint violation
execute procedure check_salary_SALGRADE();
```

A Second Trigger Example in PostgreSQL

"The total of all salaries in a department must not exceed the department's budget."

```
create function check_budget_EMP() returns trigger as '
declare
  violations  int;
begin
   if (exists (select *
               from dept D,
               (select Deptno, sum(Sal) as Salaries
                from EMP
                group by Deptno) as S
                where D.Deptno = S.DeptNo
                and D.Budget < S.Salaries
  ))  then
      raise exception ''Total of salaries in department
                        exceeds budget'';
   end if;
   return null;
end;
' language plpgsql;

create trigger check_budget_EMP
after insert or update of sal, deptno on EMP
execute procedure check_budget_EMP();
```