

Database Management System

Integrity and Security in Databases

Prof Muhammed Salman Shamsi

AIKTC

Disclaimer

- All the materials used in this presentation belongs to the respective authors mentioned in reference section.
- This presentation is to only help the students community of Mumbai University for the subject of Database Management System and is for private circulation only.
- I neither claim this material or use it for commercial purpose.
- This presentation is purely for education purpose.

Integrity 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.
- It is the mechanism to prevent invalid data entry into the table.
- Hence integrity constraints are limitations or set of rules imposed on data of database in order to keep database in consistent or correct state .
- Domain Constraints & Referential Integrity Constraints are the types of Integrity Constraints.

Domain Constraints

- Domain constraints are the most elementry form of integrity constraints.
- They test the values inserted in the database, and test queries to ensure that the comparision make sense.
- New domains can be created from the existing data types:
- **create** domain <new_domain_name> **as** <new_data_type>
  create domain Dollars **as numeric(12,2)**
  create domain Pounds **as numeric(12,2)**
- Note: we cannot assign or compare a value of type Dollars to a value of type Pounds. However we can convert type as below:
  (**cast** r.A **as** Pounds)

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.
- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S. R and S are not necessarily distinct.
- If a foreign key F in a table R refers to and matches the primary key P of table S then every value of F must either be equal to value of P or wholly NULL.
-

Cascading Actions in Referential Integrity

- **create table** *course* (
  course_id **char**(5) **primary key**,
  title **varchar**(20),
  dept_name **varchar**(20) **references** *department*
)
- **create table** *course* (
  …
  dept_name **varchar**(20),
  foreign key (*dept_name*) **references** *department*
  on delete cascade
  on update cascade,
  . . .
)
- alternative actions to cascade: **set null**, **set default**

Column Constraints and Table Constraints

If the constraints are defined along with the a column definition of a table, than they are called **column constraints**. These constraints involve only one attribute.

If more than one attribute is involved the table constraint must be used. A column constraint will not be checked if values in other columns are being updated.

If the data constraints attached to a specific column in a table references the contents of a another column in the table then they are called as **table constraints**.

Examples of different Constraints

- Not Null constraint
- Primary Key constraint
- Unique Constraint
- Default value Constraint
- Foreign Key Constraint
- Check Integrity Constraints

PK as a Column Constraint

A column constraint is usually used when the PK is a single attribute.

Constraint: Data entered in the column must be unique and not null.

CREATE TABLE *Match*
(*MatchID* INT PRIMARY KEY,
Team1 CHAR(15),
Team2 CHAR(15),
Ground CHAR(20),
Date CHAR(10),
Result CHAR(10));

PK as a Table Constraint

A table constraint is usually used when the PK is more than a single attribute.

CREATE TABLE *Bowling*
(*MID* INT,
PID INT,
NOvers INT,
Maidens INT,
NRuns INT,
NWickets INT,
PRIMARY KEY (*MID, PID*));

FK as a Column Constraint

A column constraint is usually used when the FK is a single attribute.

CREATE TABLE *Employee*
(*EmpID* NUMERIC(6) PRIMARY KEY,
Name CHAR(20),
Dept CHAR(10), REFERENCES *Department* (*DeptID*),
Address CHAR (50)
Position CHAR (20));

FK as a Table Constraint

A table constraint is usually required when the FK is more than a single attribute.

CREATE TABLE *Bowling*
(*MatchID* INT,
PID INTEGER,
NOvers INT,
Maidens INT,
NRuns INT,
NWickets INT,
PRIMARY KEY (*MID, PID*)
FOREIGN KEY (*MatchID*) REFERENCES *Match,*
FOREIGN KEY (*PID*) REFERENCES *Player*);

NULL as a Column Constraint

CREATE TABLE *Match*
(*MatchID* INT PRIMARY KEY,
Team1 CHAR(15) NOT NULL,
Team2 CHAR(15) NOT NULL,
Ground CHAR(20),
Date CHAR(10),
Result CHAR(10));

DEFAULT as a Column Constraint

CREATE TABLE *Match*

(*MatchID* INT PRIMARY KEY

Team1 CHAR(15) DEFAULT 'India',

Team2 CHAR(15),

Ground CHAR(20),

Date CHAR(10),

Result CHAR(10));

UNIQUE as a Column Constraint

A column constraint is usually used when UNIQUE is a single attribute.

CREATE TABLE *Employee*
(*EmpID* NUMBER(6) PRIMARY KEY,
Name CHAR(20),
DeptID CHAR(10),
Telephone INT UNIQUE,
Address CHAR (50),
Position CHAR (20);

UNIQUE as a Table Constraint

A table constraint is usually required when UNIQUE is more than a single attribute.

CREATE TABLE *Player*
(*PlayerID* INT PRIMARY KEY,
LName CHAR(15),
FName CHAR(15),
Country CHAR(20),
YBorn INT,
BPlace CHAR(20)
FTest INT,
UNIQUE (*LName*, *FName*));

CHECK Constraint

	Possible conditions in the CHECK clause
1	attribute A > value v
2	attribute A between value v1 and value v2
3	attribute A IN (list of values)
4	Attribute A IN subquery
5	attribute A condition C1 OR condition C2
6	attribute A condition C1 AND condition C2

CHECK as a Column Constraint

```
CREATE TABLE Player
    (PlayerID INT PRIMARY KEY,
    LName CHAR(15),
    FName CHAR(15),
    Country CHAR(20),
    YBorn INT CHECK (YBorn > 1950),
    BPlace CHAR(20),
    FTest INT);
```


CHECK as a Table Constraint

A table constraint is used when the CHECK constraint has more than a single attribute.

CREATE TABLE *Player*
(*PlayerID* INT PRIMARY KEY,
LName CHAR(15) NOT NULL,
FName CHAR(1) NOT NULL,
Country CHAR(20),
YBorn INT,
BPlace CHAR(20),
FTest INT,
CHECK (*FTest* > *YBorn* + 15));

Alternative ways to create constraints

- Syntax:
  constraint [<constraint_name>] constraint_definition;

- In create command
  create table Student(sid varchar(20), mobileno varchar(10),
  ....,
  constraint stud_pk primary key(sid),
  constraint m_unique unique(mobileno));

- In Alter command
  Alter table Student ADD CONSTRAINT check_age CHECK(age>16);

- Dropping a constraint
  Alter table Student DROP CONSTRAINT check_age;

Complex Check Clauses

- Complex check conditions can be useful when we want to ensure integrity of data, but may be costly to test.
  check (timeslot_id **in** (**select** timeslot_id **from** timeslot))
- For example, the predicate in the check clause would not only have to be evaluated when a modification is made to the section relation, but may have to be checked if a modification is made to the time slot relation because that relation is referenced in the subquery.
- **Unfortunately: subquery in check clause not supported by pretty much any database**

Assertion

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- **Domain constraints** and **referential-integrity constraints** are special forms of assertions.
- **create assertion** <assertion-name> **check** <predicate>;
  - Also rarely supported by anyone
- Two examples of such constraints are:
- For each tuple in the student relation, the value of the attribute tot_cred must equal the sum of credits of courses that the student has completed successfully.
- An instructor cannot teach in two different classrooms in a semester in the same time slot.

Assertion Example

```
create assertion credits_earned constraint check
(not exists (select ID
            from student
            where tot_cred <> (select sum(credits)
            from takes natural join course
            where student.ID= takes.ID
                and grade is not null
                and grade<> 'F' );
```

Triggers

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.

- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.
  - Specify the actions to be taken when the trigger executes.

- The above model of triggers is referred to as the **event-condition-action** model for trigger.

Need for Triggers

- Triggers can be used to implement certain integrity constraints that cannot be specified using the constraint mechanism of SQL.
- Triggers are also useful mechanisms for alerting humans or for starting certain tasks automatically when certain conditions are met.
- As an illustration, we could design a trigger that, whenever a tuple is inserted into the takes relation, updates the tuple in the student relation for the student taking the course by adding the number of credits for the course to the student’s total credits.

Trigger Syntax [MySQL]

CREATE

[**DEFINER** = { user | **CURRENT_USER** }]

TRIGGER trigger_name

trigger_time trigger_event

ON tbl_name **FOR EACH ROW**

trigger_body

trigger_time: { **BEFORE** | **AFTER** }

trigger_event: { **INSERT** | **UPDATE** | **DELETE** }

Triggering Events and Actions in SQL

Triggering event can be **insert**, **delete** or **update**

Triggers on update can be restricted to specific attributes

For example, **after update of** *takes* **on** *grade*

Values of attributes before and after an update can be referenced

referencing old row as : for deletes and updates

referencing new row as : for inserts and updates

Triggers can be activated before an event, which can serve as extra constraints. For example, convert blank grades to null.

Trigger Example IBM DB2

create trigger setnull **before update on** takes

referencing new row as nrow

for each row

when (nrow.grade = ' ')

begin atomic

    set nrow.grade = **null;**

end;

Trigger Example MySQL

```
create trigger setnull before update on takes
for each row
begin
	if new.grade ='' then
		set new.grade = null;
	end if;
end;
```

```
create trigger credits_earned after update of takes on (grade)
referencing new row as nrow
referencing 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;
```

Triggers for maintaining referential integrity

```
create trigger timeslot_check1 after insert on section
referencing new row as nrow
for each row
when (nrow.time_slot_id not in (
            select time_slot_id
                from time_slot)) /* time slot id not
                      present in time slot */
begin
    rollback
end;
```


When Not to Use Triggers

- No need to implement the on delete cascade feature of a foreign-key constraint by using a trigger, instead of use the cascade feature.
- There is no need to write trigger code for maintaining materialized views.
- Modern database systems, provide built-in facilities for database replication, making triggers unnecessary for replication in most cases.
- Triggers should be written with great care, since a trigger error detected at runtime causes the failure of the action statement that set off the trigger. Furthermore, the action of one trigger can set off another trigger. In the worst case, this could even lead to an infinite chain of triggering.
- Many trigger applications can be substituted by appropriate use of stored procedures

Security

- Security is a protection from malicious attempts to steal or modify data. The security should be provided at following levels:
- 1) **Database system level**. (user acess only required data)
- 2) **Operating system level**. (super user)
- 3) **Network level**. (encryption,eavesdropping,masquerading)
- 4) **Physical level**.
- 5) **Human level**. (user training)

Authorization

Forms of authorization on parts of the database:

Read - allows reading, but not modification of data.
Insert - allows insertion of new data, but not modification of existing data.
Update - allows modification, but not deletion of data.
Delete - allows deletion of data.

Forms of authorization to modify the database schema

Index - allows creation and deletion of indices.
Resources - allows creation of new relations.
Alteration - allows addition or deletion of attributes in a relation.
Drop - allows deletion of relations.

Authorization Specification in SQL

The grant statement is used to confer authorization

grant <privilege list> **on** <relation name or view name> **to** <user list>

<user list> is:

a user-id

public, which allows all valid users the privilege granted

A role (more on this later)

Granting a privilege on a view does not imply granting any privileges on the underlying relations.

Privileges in SQL

select: allows read access to relation,or the ability to query using the view

Example: grant users U_1, U_2, and U_3 **select** authorization on the *instructor* relation:

grant select on *instructor* **to** U_1, U_2, U_3

insert: the ability to insert tuples
update: the ability to update using the SQL update statement
delete: the ability to delete tuples.
all privileges: used as a short form for all the allowable privileges

Revoking Authorization in SQL

The revoke statement is used to revoke authorization.

revoke <privilege list>
on <relation name or view name> **from** <user list>

Example:

revoke select on *branch* **from** U_1, U_2, U_3

<privilege-list> may be **all** to revoke all privileges the revokee may hold.

If <revokee-list> includes **public,** all users lose the privilege except those granted it explicitly.

If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.

All privileges that depend on the privilege being revoked are also revoked.

Roles

- **create role** instructor;
- **grant** *instructor* **to Amit;**
- Privileges can be granted to roles:
  - **grant select on** *takes* **to** *instructor*;
- Roles can be granted to users, as well as to other roles
  - **create role** *teaching_assistant*
  - **grant** *teaching_assistant* **to** *instructor*;
    - *Instructor* inherits all privileges of *teaching_assistant*
- Chain of roles
  - **create role** *dean*;
  - **grant** *instructor* **to** *dean*;
  - **grant** *dean* **to** Satoshi;

Limitations of SQL Authorization

- SQL does not support authorization at a tuple level.
- All end users of an application may be mapped to a single database user.
- The task of authorization in above cases falls on the application program, with no support from SQL:

  Benefits: Fine grained authorizations implemented by applications

  Drawback: Authorization loopholes are created which becomes difficult to find due to large amount of application code

References

- Database Management System, G.K Gupta, Tata McGraw Hill
- Database System Concepts, Korth, Sudarshan et. al., Tata McGraw Hill