

SQL: Views, Triggers

CS348

Instructor: Sujaya Maiyya

Views

- A **view** is like a “virtual” table
 - Defined by a query, which describes **how to compute the view contents on the fly**
 - Stored as a query by DBMS instead of query contents
 - Can be used in queries just like a regular table

```
CREATE VIEW PopGroup AS
  SELECT * FROM User
  WHERE uid IN (SELECT uid
                FROM Member
                WHERE gid = 'popgroup');
```



```
SELECT AVG(pop)
FROM (SELECT * FROM User
      WHERE uid IN
      (SELECT uid FROM Member
        WHERE gid = 'popgroup'))
AS popGroup;
```

```
SELECT AVG(pop) FROM PopGroup;
```



```
SELECT MIN(pop) FROM PopGroup;
```

```
SELECT ... FROM PopGroup;
```

```
DROP VIEW popGroup;
```

Why use views?

- To **hide complexity** from users
- To **hide data** from users
- **Logical** data independence
- To provide a **uniform interface**

Modifying views

- Does it even make sense, since views are virtual?
- It does make sense if we want users to really see views as tables
- Goal: **modify the base tables** such that the modification would **appear to have been accomplished on the view**

A simple case

```
CREATE VIEW UserPop AS  
    SELECT uid, pop FROM User;
```

```
DELETE FROM UserPop WHERE uid = 123;
```

translates to:

```
DELETE FROM User WHERE uid = 123;
```

An impossible case

```
CREATE VIEW PopularUser AS  
  SELECT uid, pop FROM User  
  WHERE pop >= 0.8;
```

```
INSERT INTO PopularUser VALUES(987, 0.3);
```

- No matter what we do on *User*, the inserted row will not be in *PopularUser*

A case with too many possibilities

```
CREATE VIEW AveragePop(pop) AS  
SELECT AVG(pop) FROM User;
```

Renamed
column

```
UPDATE AveragePop SET pop = 0.5;
```

- Set everybody's *pop* to 0.5?
- Adjust everybody's *pop* by the same amount?
- Just lower one user's *pop*?

SQL92 updateable views

- More or less just single-table selection queries
 - No join
 - No aggregation or group by
 - No subqueries
 - Attributes not listed in SELECT must be nullable
- Arguably somewhat restrictive
- Still might get it wrong in some cases
 - See the slide titled “An impossible case”
 - Adding **WITH CHECK OPTION** to the end of the view definition will make DBMS reject such modifications

Materialized views

- Some systems allow view relations to be stored in db
 - If the actual relations used in the view definition change, the view is kept up-to-date
- Such views are called **materialized views**
- Used to enhance performance: avoid recomputing view each time
- **View maintenance:** updating the materialized view upon base table changes
 - Immediately or lazily, up to the DBMS

Exercises

Consider this db instance:

User

<i>uid</i>	<i>name</i>	<i>age</i>	<i>pop</i>
142	Bart	10	0.9
123	Milhouse	10	0.2
857	Lisa	8	0.7
456	Ralph	7	0.3

Member

<i>uid</i>	<i>gid</i>
857	dps
123	gov
857	abc
857	gov
456	abc
456	gov

- What is the output of these queries?

```
CREATE VIEW ageGroups(age,cnt) AS  
  (SELECT age, COUNT(*) FROM User GROUP BY age)
```

```
SELECT * FROM ageGroups;
```

```
SELECT age FROM ageGroups  
WHERE cnt = (SELECT MAX(cnt) FROM ageGroups);
```

Exercises

User (uid int, name string, age int, pop float)
Group (gid string, name string)
Member (uid int, gid string)

- Create a view that captures all the users, groups, and their membership information in one master table. The view should show information of users who do not belong to a group and groups that have no members.
- From this view, find uid and names of users who belong to at least two groups.

SQL

- Views
- Triggers

Recall “referential integrity”

Example: *Member.uid* references *User.uid*

- Delete or update a *User* row whose *uid* is referenced by some *Member* row
 - Multiple Options (in SQL)

User			Member	
uid	name	...	uid	gid
142	Bart	...	142	dps
123	Milhouse	...	123	gov
857	Lisa	...	857	abc
456	Ralph	...	857	gov
789	Nelson	...	456	abc
...	...	...	456	gov
			...	...

Option 1: Reject

```
CREATE TABLE Member
(uid INT NOT NULL
REFERENCES User(uid)
ON DELETE CASCADE,
....);
```

Option 2: Cascade
(ripple changes to all referring rows)

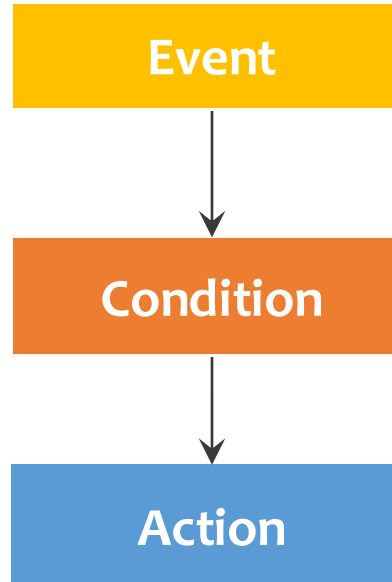
Can we generalize it?

Referential constraints

Delete/update a
User row

Whether its uid is
referenced by some
Member row

If yes: reject/ delete
cascade/null



Data Monitoring

Some user's
popularity is updated

Whether the user is a
member of "Pop group"
and pop drops below 0.5

If yes: kick that user out
of Pop group!

Triggers

- A **trigger** is an event-condition-action (ECA) rule
 - When **event** occurs, test **condition**; if condition is satisfied, execute **action**

```
CREATE TRIGGER PickyPopGroup
AFTER UPDATE OF pop ON User
REFERENCING NEW ROW AS newUser
FOR EACH ROW
    WHEN (newUser.pop < 0.5)
        AND (newUser.uid IN (SELECT uid
                              FROM Member
                              WHERE gid = 'popgroup'))
    DELETE FROM Member
    WHERE uid = newUser.uid AND gid = 'popgroup'
```

Diagram illustrating the ECA rule components in the SQL trigger code:

- Event**: UPDATE OF pop ON User
- Transition variable**: newUser
- Condition**: WHEN (newUser.pop < 0.5) AND (newUser.uid IN (SELECT uid FROM Member WHERE gid = 'popgroup'))
- Action**: DELETE FROM Member WHERE uid = newUser.uid AND gid = 'popgroup'

Trigger option 1 – possible events

- Possible events include:
 - **INSERT ON** *table*; **DELETE ON** *table*; **UPDATE** [**OF column**]
ON *table*

```
CREATE TRIGGER PickyPopGroup
AFTER UPDATE OF pop ON User
REFERENCING NEW ROW AS newUser
FOR EACH ROW
    WHEN (newUser.pop < 0.5)
        AND (newUser.uid IN (SELECT uid
                              FROM Member
                              WHERE gid = 'popgroup'))
        DELETE FROM Member
        WHERE uid = newUser.uid AND gid = 'popgroup';
```

Diagram illustrating the trigger definition with annotations:

- Event**: Points to the trigger event `UPDATE OF pop ON User`.
- Condition**: Points to the `WHEN` clause condition `(newUser.pop < 0.5) AND (newUser.uid IN (SELECT uid FROM Member WHERE gid = 'popgroup'))`.
- Action**: Points to the `DELETE FROM Member WHERE uid = newUser.uid AND gid = 'popgroup';` statement.

Trigger option 2 – timing

- Timing—action can be executed:
 - **AFTER** or **BEFORE** the triggering event
 - **INSTEAD OF** the triggering event on views (not covered in this course)

```
CREATE TRIGGER NoFountainOfYouth
BEFORE UPDATE OF age ON User
REFERENCING OLD ROW AS o, NEW ROW AS n
FOR EACH ROW
  WHEN (n.age < o.age)
    SET n.age = o.age;
```

The diagram illustrates the components of the SQL trigger code. Three callout boxes with orange borders and lines pointing to specific parts of the code are present: a box labeled 'Event' points to 'UPDATE OF age ON User'; a box labeled 'Condition' points to the 'WHEN (n.age < o.age)' clause; and a box labeled 'Action' points to the 'SET n.age = o.age;' statement.

Trigger option 3 – granularity

- Granularity—trigger can be activated:
 - **FOR EACH ROW** modified

```
CREATE TRIGGER PickyPopGroup
AFTER UPDATE OF pop ON User
REFERENCING NEW ROW AS newUser
FOR EACH ROW
  WHEN (newUser.pop < 0.5)
    AND (newUser.uid IN (SELECT uid
                        FROM Member
                        WHERE gid = 'popgroup'))
  DELETE FROM Member
  WHERE uid = newUser.uid AND gid = 'popgroup';
```

Diagram illustrating the trigger options in the SQL code:

- Event**: Points to the trigger event `UPDATE OF pop ON User`.
- Condition**: Points to the `WHEN` clause condition `(newUser.pop < 0.5) AND (newUser.uid IN (SELECT uid FROM Member WHERE gid = 'popgroup'))`.
- Action**: Points to the `DELETE FROM Member WHERE uid = newUser.uid AND gid = 'popgroup';` statement.

Trigger option 3 – granularity

- Granularity—trigger can be activated:
 - **FOR EACH ROW** modified
 - **FOR EACH STATEMENT** that performs modification

```
CREATE TRIGGER PickyPopGroup2
AFTER UPDATE OF pop ON User
REFERENCING NEW TABLE AS newUsers
FOR EACH STATEMENT
DELETE FROM Member
WHERE gid = 'popgroup'
AND uid IN (SELECT uid
FROM newUsers
WHERE pop < 0.5);
```

Event

Transition table:
contains all the
affected rows

Condition
& Action

Trigger option 3 – granularity

- Granularity—trigger can be activated:
 - **FOR EACH ROW** modified
 - **FOR EACH STATEMENT** that performs modification

```
CREATE TRIGGER PickyPopGroup2
AFTER UPDATE OF pop ON User
REFERENCING NEW TABLE AS newUsers
FOR EACH STATEMENT
    DELETE FROM Member
        WHERE gid = 'popgroup'
        AND uid IN (SELECT uid
                    FROM newUsers
                    WHERE pop < 0.5);
```

Transition table:
contains all the
affected rows

Can only be used
with **AFTER**
triggers

Statement- vs. row-level triggers

- Simple row-level triggers are easier to implement
 - Statement-level triggers: require significant amount of state to be maintained in OLD TABLE and NEW TABLE
- However, in some cases a row-level trigger may be less efficient
 - E.g., 4B rows and a trigger may affect 10% of the rows. Recording an action for 4 Million rows, one at a time, is not feasible due to resource constraints.
- Certain triggers are only possible at statement level
 - E.g., ??

Certain triggers are only possible at statement level

```
CREATE TRIGGER MaintainAvgPop
AFTER UPDATE OF pop ON User
REFERENCING NEW TABLE AS newUsers
          OLD TABLE AS oldUsers
FOR EACH STATEMENT
  WHEN (0.5 > (SELECT AVG(pop) FROM User))
  BEGIN
    DELETE FROM User WHERE uid IN (SELECT uid
    FROM newUsers)
    INSERT INTO User (SELECT * FROM oldUsers)
  END
```

The diagram illustrates the components of the trigger statement:

- Event:** Points to the `UPDATE OF pop ON User` clause.
- Transition tables:** Points to the `REFERENCING NEW TABLE AS newUsers OLD TABLE AS oldUsers` clause.
- Condition:** Points to the `WHEN (0.5 > (SELECT AVG(pop) FROM User))` clause.
- Action:** Points to the `DELETE FROM User WHERE uid IN (SELECT uid FROM newUsers) INSERT INTO User (SELECT * FROM oldUsers)` block.

System issues

- Recursive firing of triggers
 - Action of one trigger causes another trigger to fire
 - Can get into an infinite loop
- Interaction with constraints (tricky to get right!)
 - When to check if a triggering event violates constraints?
 - After a BEFORE trigger
 - Before an AFTER trigger
 - (Typical but db dependent)
- Best to avoid when alternatives exist

Exercises

User (uid int, name string, age int, pop float)
Group (gid string, name string)
Member (uid int, gid string)

- If a user with $pop > 0.5$ is added to the User table, they must automatically belong to the 'popgroup'. Create a trigger to achieve this behavior.
 - Assume 'popgroup' already exists in the Group table
- Write the trigger once using FOR EACH ROW and once using FOR EACH STATEMENT

Exercises– FOR EACH ROW

- If a user with $pop > 0.5$ is added to the User table, they must automatically belong to the 'popgroup'. Create a trigger to achieve this behavior.

```
CREATE TRIGGER AddToPopgroup
AFTER INSERT ON User
REFERENCING NEW ROW AS newUser
FOR EACH ROW
  WHEN (newUser.pop > 0.5)
    INSERT INTO Member
    VALUES (newUser.uid, 'popgroup')
```

The diagram illustrates the components of the SQL trigger code. Callouts point to specific parts of the code:

- Event:** Points to `AFTER INSERT ON User`.
- Transition variable:** Points to `newUser` in the `REFERENCING` clause.
- Condition:** Points to `WHEN (newUser.pop > 0.5)`.
- Action:** Points to the `INSERT INTO Member VALUES (newUser.uid, 'popgroup')` statement.

Exercises – FOR EACH STATEMENT

- If a user with $pop > 0.5$ is added to the User table, they must automatically belong to the 'popgroup'. Create a trigger to achieve this behavior.

```
CREATE TRIGGER AddToPopgroup
AFTER INSERT ON User
REFERENCING NEW TABLE AS newUsers
FOR EACH STATEMENT
    INSERT INTO Member
    (SELECT uid, 'popgroup' FROM newUsers n
     WHERE n.pop > 0.5)
```

The diagram illustrates the components of the SQL trigger statement:

- Event**: Points to the `INSERT ON User` part of the statement.
- Transition variable**: Points to the `newUsers` alias in the `REFERENCING` clause.
- Condition & Action**: Points to the `FOR EACH STATEMENT` clause and the `INSERT INTO Member` statement.

Summary

- Triggers
- View
- Optional slides on *programming + SQL* and *Recursion in SQL*
- Next week: entity relationship modeling