

SQL: Triggers, Views, Indexes

CS348 Spring 2024

Instructor: Sujaya Maiyya

Sections: **002 & 003 only**

Announcements

- Assignment 1: Due June 4th
 - SQL questions on Marmoset
 - Other questions on Crowdmark

Triggers - Recap

- 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'
```

The diagram illustrates the components of the SQL trigger code. It features four callout boxes with lines pointing to specific parts of the code:

- Event:** Points to the text "UPDATE OF pop ON User".
- Transition variable:** Points to the underlined text "NEW ROW AS newUser".
- Condition:** Points to the text "WHEN (newUser.pop < 0.5) AND (newUser.uid IN (SELECT uid FROM Member WHERE gid = 'popgroup'))".
- Action:** Points to the text "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';
```

Event

Condition

Action

Trigger option 2 – timing

- Timing—action can be executed:
 - **AFTER** or **BEFORE** the triggering event
 - **INSTEAD OF** the triggering event on views (more later)

```
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 a SQL trigger. It features a black background with white and colored text. Three callout boxes with orange borders and lines pointing to specific parts of the SQL code are present: 'Event' points to 'UPDATE OF age ON User', 'Condition' points to '(n.age < o.age)', and 'Action' points to 'SET n.age = o.age;'. The words 'OLD ROW AS o' and 'NEW ROW AS n' are underlined in blue.

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';
```

Event

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);
```

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

Transition variables/tables

- **OLD ROW**: the modified row before the triggering event
- **NEW ROW**: the modified row after the triggering event
- **OLD TABLE**: a read-only table containing all old rows modified by the triggering event
- **NEW TABLE**: a table containing all modified rows after the triggering event

Event	Row	Statement
Delete	old r; old t	old t
Insert	new r; new t	new t
Update	old/new r; old/new t	old/new t

AFTER Trigger

Event	Row	Statement
Update	old/new r	-
Insert	new r	-
Delete	old r	-

BEFORE Trigger

In class 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

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

In class 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 code. Three callout boxes are connected to the code by lines:

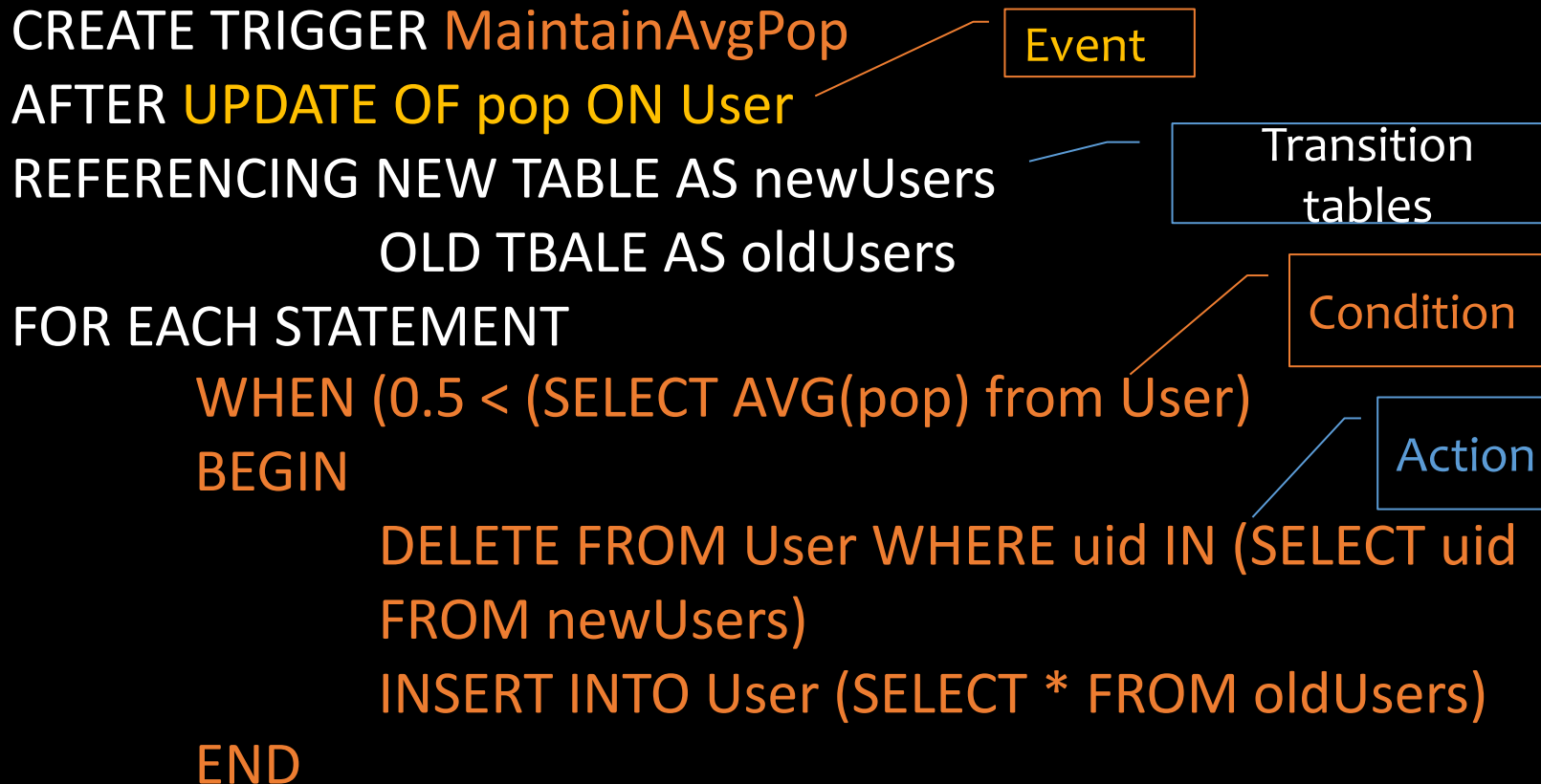
- Event:** A box containing the text "Event" with a line pointing to the trigger name "AddToPopgroup".
- Transition variable:** A box containing the text "Transition variable" with a line pointing to the table alias "newUsers".
- Condition & Action:** A box containing the text "Condition & Action" with a line pointing to the `INSERT INTO Member` statement.

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, may not be 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 SQL trigger code:

- Event:** `UPDATE OF pop ON User`
- Transition tables:** `newUsers` and `oldUsers`
- Condition:** `0.5 < (SELECT AVG(pop) from User)`
- Action:** `DELETE FROM User WHERE uid IN (SELECT uid FROM newUsers)` and `INSERT INTO User (SELECT * FROM oldUsers)`

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
 - (based on db2, other DBMS may differ)
- Best to avoid when simpler alternatives are sufficient

SQL features covered so far

- Basic SQL
- Intermediate SQL
 - Triggers
 - Views

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');
```

Base tables

```
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

INSTEAD OF triggers for views

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

```
CREATE TRIGGER AdjustAveragePop  
INSTEAD OF UPDATE ON AveragePop  
REFERENCING OLD ROW AS o,  
NEW ROW AS n  
FOR EACH ROW  
UPDATE User  
SET pop = pop + (n.pop-o.pop);
```

- What does this trigger do?

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


INSTEAD OF triggers for views

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

```
CREATE TRIGGER AdjustAveragePop  
INSTEAD OF UPDATE ON AveragePop  
REFERENCING OLD ROW AS o,  
NEW ROW AS n  
FOR EACH ROW  
UPDATE User  
SET pop = pop + (n.pop-o.pop);
```

0.5

0.4

User

...	pop	...
	0.4	+0.1
	0.4	+0.1
	0.5	+0.1
	0.3	+0.1

- What does this trigger do?

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

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

In class 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);
```

In class exercises

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

- Assume there is a CHECK constraint on User table s.t. ($age > 0$ and $age < 140$)

```
CREATE VIEW youngUsers AS  
  (SELECT * FROM User WHERE age < 25) WITH CHECK OPTION;
```

- What happens to the following statements?

```
INSERT INTO youngUsers VALUES (835, 'Alex', 30, 0.2);
```

```
INSERT INTO youngUsers VALUES (923, 'James', 150, 0.3);
```

SQL features covered so far

- Basic SQL
- Intermediate SQL
 - Triggers
 - Views
 - Indexes

Motivating examples of using indexes

```
SELECT * FROM User WHERE name = 'Bart';
```

- Can we go “directly” to rows with *name*='Bart' instead of scanning the entire table?
 - index on *User.name*

```
SELECT * FROM User, Member  
WHERE User.uid = Member.uid AND Member.gid = 'popgroup';
```

- Can we find relevant *Member* rows “directly”?
 - index on *Member.gid*
- For each relevant *Member* row, can we “directly” look up *User* rows with matching *Member.uid*
 - index on *User.uid*

Indexes

- An **index** is an auxiliary persistent data structure that helps with efficient searches
 - Search tree (e.g., B⁺-tree), lookup table (e.g., hash table), etc.
 - ☞ More on indexes later in this course!
- **CREATE [UNIQUE] INDEX *indexname* ON *tablename*(*columnname*₁, ..., *columnname*_{*n*});**
 - With UNIQUE, the DBMS will also enforce that {*columnname*₁, ..., *columnname*_{*n*}} is a key of *tablename*
- **DROP INDEX *indexname*;**
- Typically, the DBMS will automatically create indexes for PRIMARY KEY and UNIQUE constraint declarations

Indexes

- An index on $R.A$ can speed up accesses of the form
 - $R.A = value$
 - $R.A > value$ (sometimes; depending on the index type)
- An index on $(R.A_1, \dots, R.A_n)$ can speed up
 - $R.A_1 = value_1 \wedge \dots \wedge R.A_n = value_n$
 - $(R.A_1, \dots, R.A_n) > (value_1, \dots, value_n)$ (again depends)

Questions (lecture 12):

- ☞ Ordering of index columns is important—is an index on $(R.A, R.B)$ equivalent to one on $(R.B, R.A)$?
- ☞ How about an index on $R.A$ plus another on $R.B$?
- ☞ More indexes = better performance?

Summary

- Triggers
 - View
 - Indexes
-
- Next: Programming and recursion