SFU

Database Systems I

CMPT 354 Summer 2024

Zhengjie Miao

7 June 2024

Announcements (Fri. June 7)

- Midterm I (Wed. June 12)
  - Sample released
    - Just an example of the format and style
  - Exam will be 75 min (not 80 min!)
  - Covers Lec1-Lec8, including the part taught on Friday, June 7
  - Open-book, open-notes, but
    - no collaboration
    - no laptop/phones/other electronics
  - Please arrive early, the exam starts exactly at 3:35 and ends exactly at 4:50
    - Bring your SFU id

SQL features covered so far

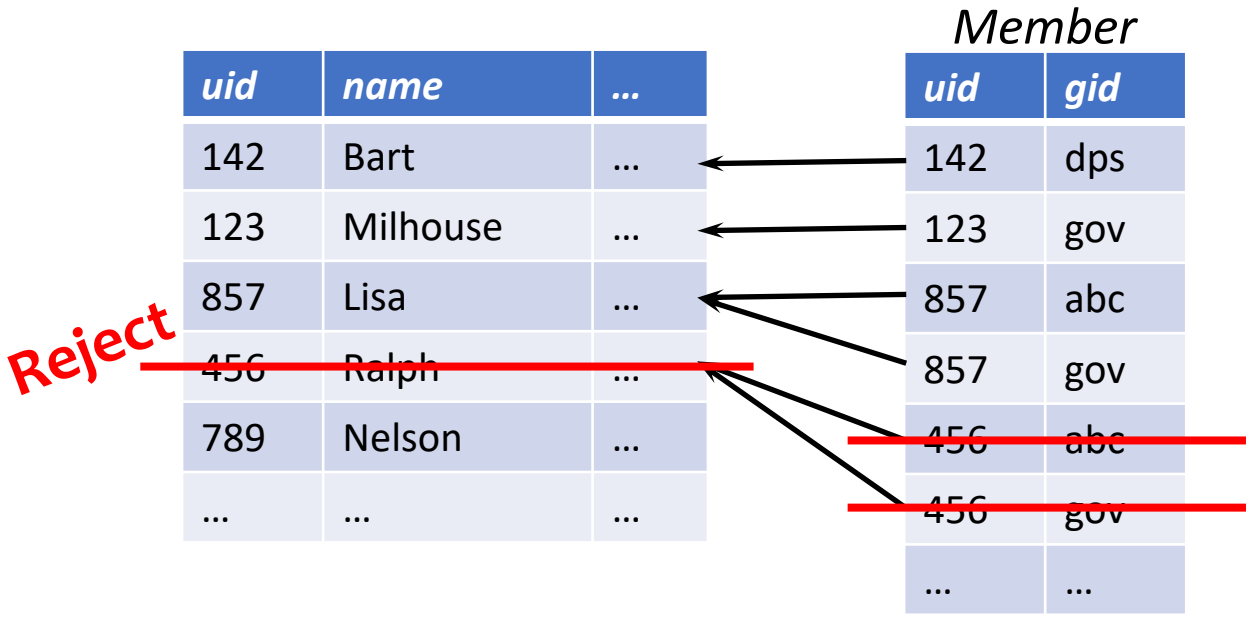
- Query
  - `SELECT-FROM-WHERE` statements
  - Set and bag operations
  - Table expressions, subqueries
  - Aggregation and grouping
  - Ordering
  - `NULL` and Outerjoins
- Modification & Constraints

☞Today : triggers and views

Recap: Enforcing referential integrity

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

- Delete or update a *User* row whose *uid* is referenced by some *Member* row
  - Option 1: Reject
  - Option 2: Cascade --- ripple changes to all referring rows




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

“Active” data

- Constraint enforcement: When an operation violates a constraint, abort the operation or try to “fix” data
  - Example: enforcing referential integrity constraints
  - Generalize to arbitrary constraints?
- Data monitoring: When something happens to the data, automatically execute some action
  - Example: When price rises above $20 per share, sell
  - Example: When enrollment is at the limit and more students try to register, email the instructor

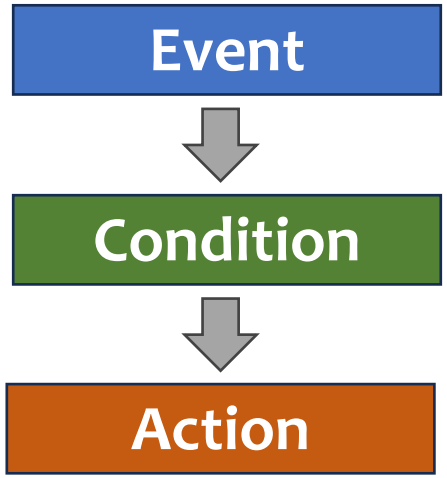
Triggers

- A trigger is an event-condition-action (ECA) rule
  - When event occurs, test condition; if condition is satisfied, execute action
  - Different DBMS support different syntax, but concepts remain the same
    - E.g., PostgreSQL syntax ≠ what we'll present here
- Example:
  - Event: some user's popularity is updated
  - Condition: the user is a member of cks ("Cool Kids") and *pop* drops below 0.5
  - Action: kick that user out of cks

https://static.wikia.nocookie.net/simpsons/images/5/55/Cool_kid.JPG

Triggers

- A trigger is an event-condition-action (ECA) rule
  - When event occurs, test condition; if condition is satisfied, execute action
  - Different DBMS support different syntax, but concepts remain the same
    - E.g., PostgreSQL syntax ≠ what we'll present here



Delete/update a row in User

Whether its uid is referenced by some row in Member

If Yes: reject/delete/ cascade/NULL

Referential constraints

Some user's popularity is updated

The user is a member of cks ("Cool Kids") and *pop* drops below 0.5

If Yes: kick that user out of cks

Data Monitoring

Trigger example

Review the syntax for a trigger

```
CREATE TRIGGER PickyCKS
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 = 'cks'))
DELETE FROM Member
WHERE uid = newUser.uid AND gid = 'cks';
```

Event

Condition

Action

Trigger option 1

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

```
CREATE TRIGGER PickyCKS
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 = 'cks'))
DELETE FROM Member
WHERE uid = newUser.uid AND gid = 'cks';
```

Event

Condition

Action

Trigger option 2

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

```
CREATE TRIGGER PickyCKS
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 = 'cks'))
DELETE FROM Member
WHERE uid = newUser.uid AND gid = 'cks';
```

Event

Condition

Action

Trigger option 2

- 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;
```

Event

Condition

Action

- BEFORE triggers are often used to "condition" data
- Another option is to raise an error in the trigger body to abort the transaction that caused the trigger to fire

Trigger option 3 (Row-level)

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

```
CREATE TRIGGER PickyCKS
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 = 'cks'))
DELETE FROM Member
WHERE uid = newUser.uid AND gid = 'cks';
```

Event

Condition

Action

Trigger option 3 (Statement-level)

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

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

Event

Condition & Action

Transition variables

- `OLD ROW`: the modified row before the triggering event
- `NEW ROW`: the modified row after the triggering event
- `OLD TABLE`: a hypothetical read-only table containing all rows to be modified before the triggering event
- `NEW TABLE`: a hypothetical table containing all modified rows after the triggering event
- Not all of them make sense all the time, e.g.
  - `BEFORE/AFTER INSERT` statement-level triggers
    - Can use only `NEW TABLE`
  - `BEFORE/AFTER UPDATE` row-level triggers
    - Can use only `OLD ROW` and `NEW ROW`
  - `BEFORE/AFTER DELETE` row-level triggers
    - Can use only `OLD ROW`
  - etc.

Why hypothetical tables?
Is the new table hypothetical since we may reject it?

Statement- vs. row-level triggers

Review statement and row-level triggers

Why are both needed?

- Certain triggers are only possible at statement level
  - If the number of users inserted by this statement exceeds 100 and their average age is below 13, then …
- 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, a row-level trigger gets fired for each row, so complex row-level triggers may be less efficient for statements that modify many rows

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 do we check if a triggering event violates constraints?
    - After a BEFORE trigger (so the trigger can fix a potential violation)
    - Before an AFTER trigger
  - AFTER triggers also see the effects of, say, cascaded deletes caused by referential integrity constraint violations

Views

- A view is like a "virtual" table
  - Defined by a query, which describes how to compute the view contents on the fly
  - DBMS stores the view definition query instead of view contents
  - Can be used in queries just like a regular table
- In contrast to temporary tables defined by WITH (visible only in the same statement), views are part of the schema and visible to all statements

Creating and dropping views

- Example: members of Cool Kids
  - ```
    CREATE VIEW CoolKids AS
    SELECT * FROM User
    WHERE uid IN (SELECT uid FROM Member
                  WHERE gid = 'cks');
    ```
  - Tables used in defining a view are called “base tables”
    - E.g., *User* and *Member* above
- To drop a view
  - ```
    DROP VIEW CoolKids;
    ```

Using views in queries

- Example: find the average popularity of members in Cool Kids
  - `SELECT AVG(pop) FROM CoolKids;`
  - To process the query, replace the reference to the view by its definition
  - ```
    SELECT AVG(pop)
    FROM (SELECT * FROM User
          WHERE uid IN
            (SELECT uid FROM Member
             WHERE gid = 'cks'))
         AS CoolKids;
    ```
    ```

Why use views?

- To hide data from users
- To hide complexity from users
- Logical data independence
  - If applications deal with views, we can change the underlying schema without affecting applications
  - Recall physical data independence: change the physical organization of data without affecting applications
- To provide a uniform interface for different implementations or sources

☞Real database applications use tons of views

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;
```

- Note that you can rename columns in view definition

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

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

SQL92 updateable views

- More or less just single-table selection queries
  - No join
  - No aggregation
  - No subqueries

- 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?

...	*pop*
	0.9
	0.7
	0.3
	0.5

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

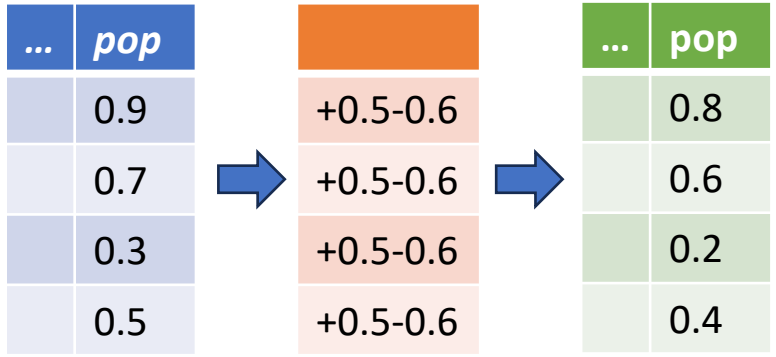
INSTEAD OF triggers for views

```
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);
```

0.6

0.5



- What does this trigger do?

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

SQL features covered so far

- Query
- Modification
- Constraints
- Triggers
- Views