Lecture 8: SQL (Advanced)

CS348 Spring 2025:
Introduction to Database Management

Instructor: **Xiao Hu**

Sections: 001, 002, 003

Announcement

- Assignment 1 Due on Jun 1

SQL features covered so far

- Basic topics:
  - Data-definition language (DDL)
  - Data-manipulation language (DML)
  - Constraints
- Advanced topics:
  - View

Post-Lecture: Views and Table Subquery

- By Default, a view is simply a stored query and the query result is NOT physically stored separately
  - But, systems can choose to materialize view (i.e., physically store the query results)

- A table subquery is a temporary table that physically stores data and it only exists in the duration of the whole query that uses it

POST-Lecture: View

Given any DML that attempts to modify view but violate the view's filter:

- If WITH CHECK OPTION: reject outright
- If WITH CHECK OPTION is not specified: it is possible to "sneak" rows into the base table through the view -- these rows simply won't appear in the view
  - However, such rows can only be inserted into the base table if they still satisfy the table's constraints

POST-Lecture: View

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 such that age is above 0 and below 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);
```

  - (835, 'Alex', 30, 0.2) will not be inserted into User, and it also will not appear in youngUser
  - Same for (923, 'James', 150, 0.3)

POST-Lecture: View

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 such that age is above 0 and below 140

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

- What happens to the following statements?

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

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

  - (835, 'Alex', 30, 0.2) can be inserted into User table but it will not appear in youngUser
  - (923, 'James', 150, 0.3) will not be inserted into User (since there is a CHECK constraint on the User table itself) and it also will not appear in youngUser

What is next?

- Basic topics:
  - Data-definition language (DDL)
  - Data-manipulation language (DML)
  - Constraints
- Advanced topics:
  - View
  - Trigger

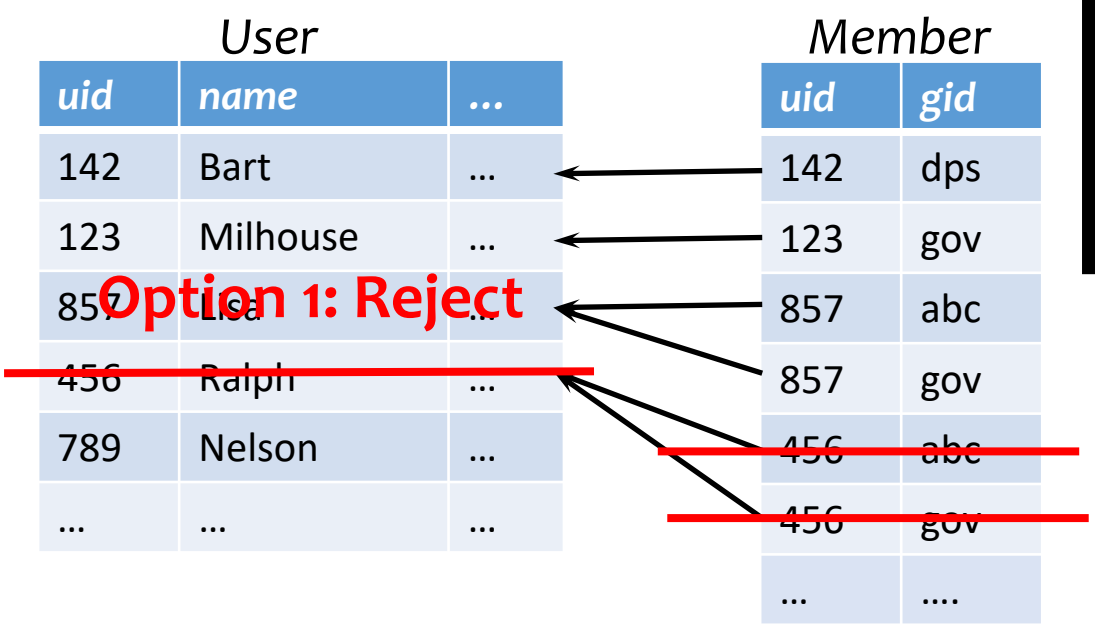
(Recap) Referential Integrity

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

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

Option 3: Set NULL

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



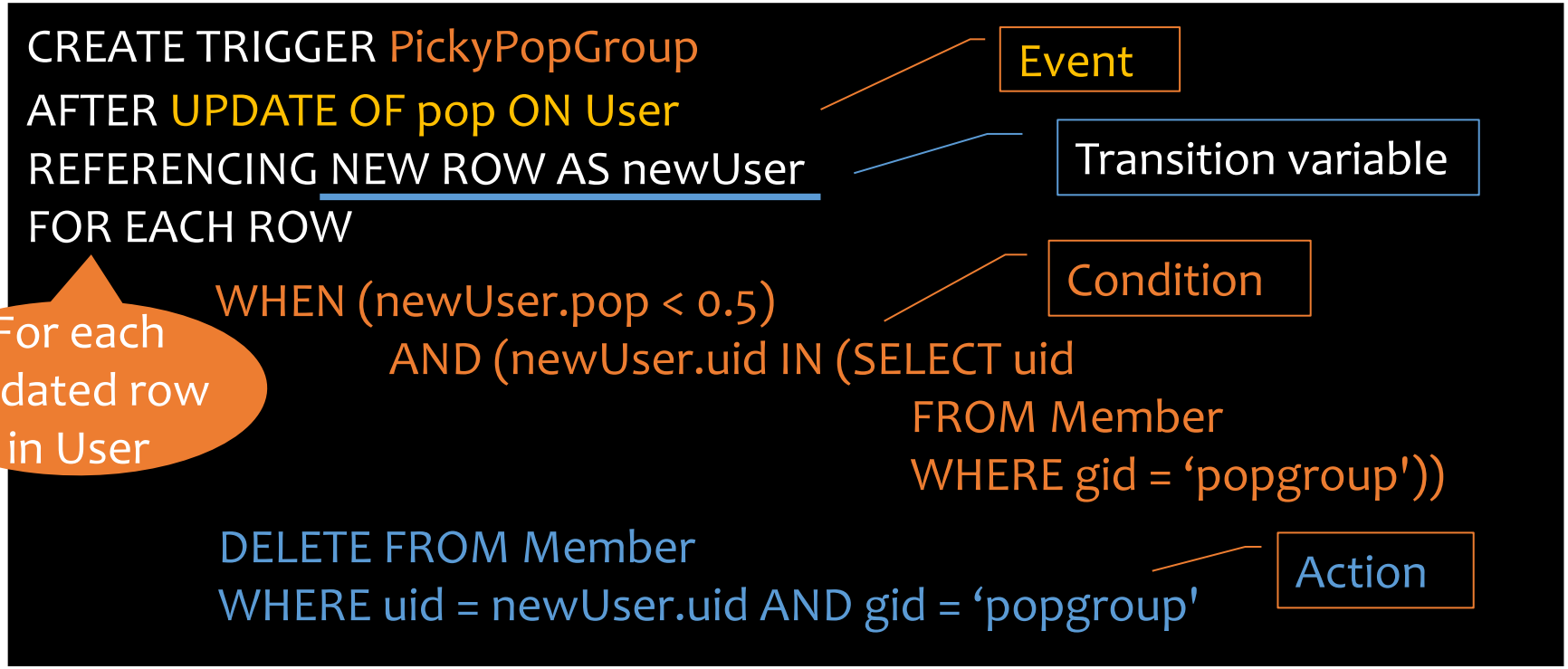
Option 2: Cascade
(ripple changes to all referring rows)

Can we generalize it?

Referential constraints		General constraints
Delete/update a User row	Event	Some user's popularity is updated
Whether its uid is referenced by some Member row	Condition	Whether the user is a member of "popgroup" and *pop* drops below 0.5
If yes: Reject/ Delete cascade/Null	Action	If yes: kick that user out of popgroup!

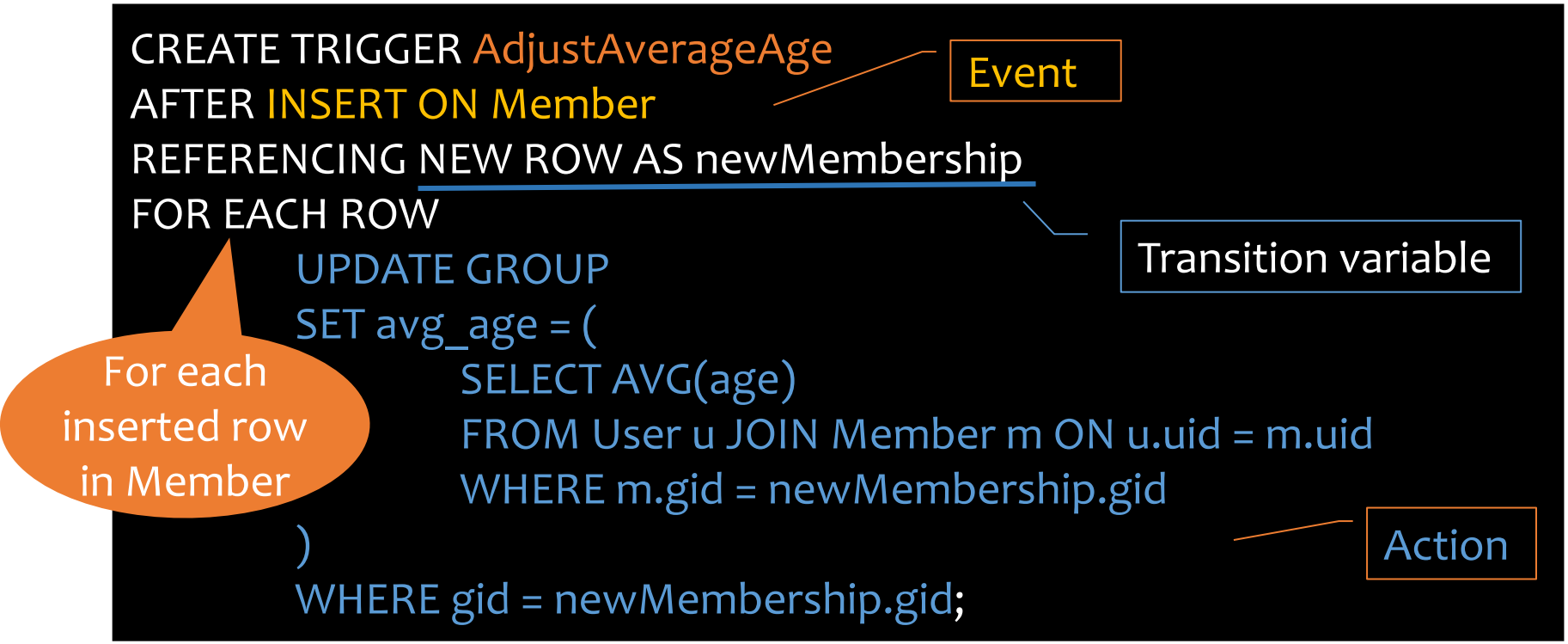
Triggers

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



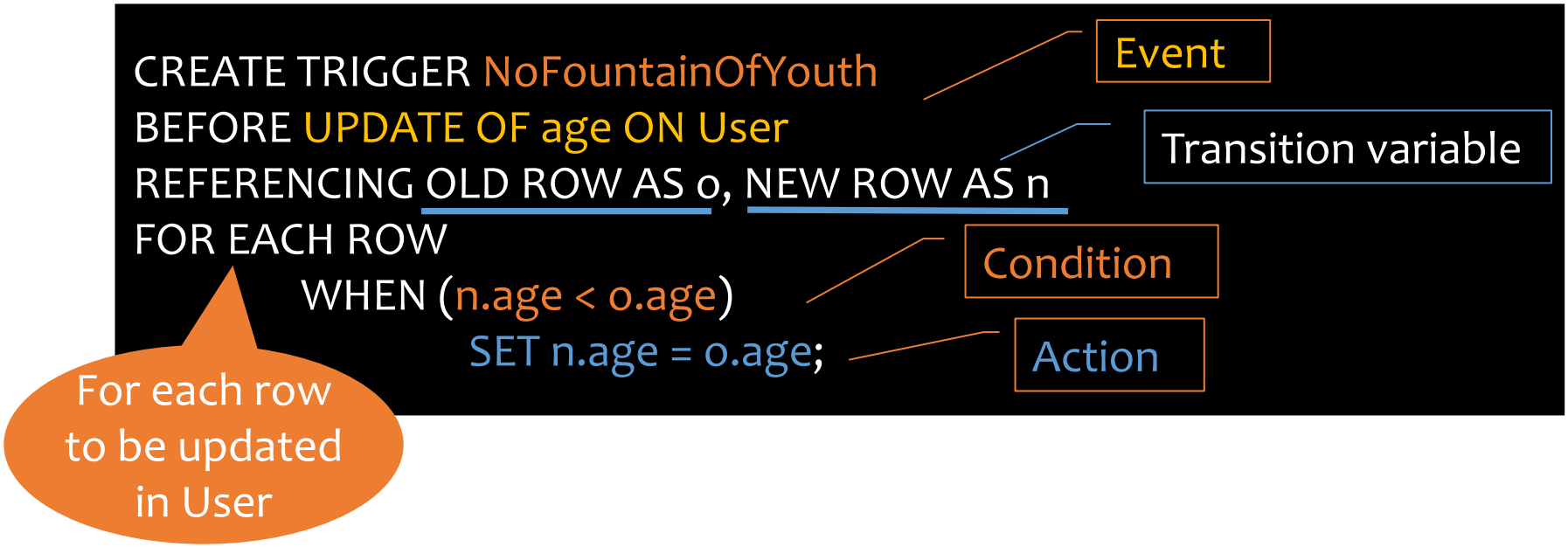
Trigger option 1 – possible events

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



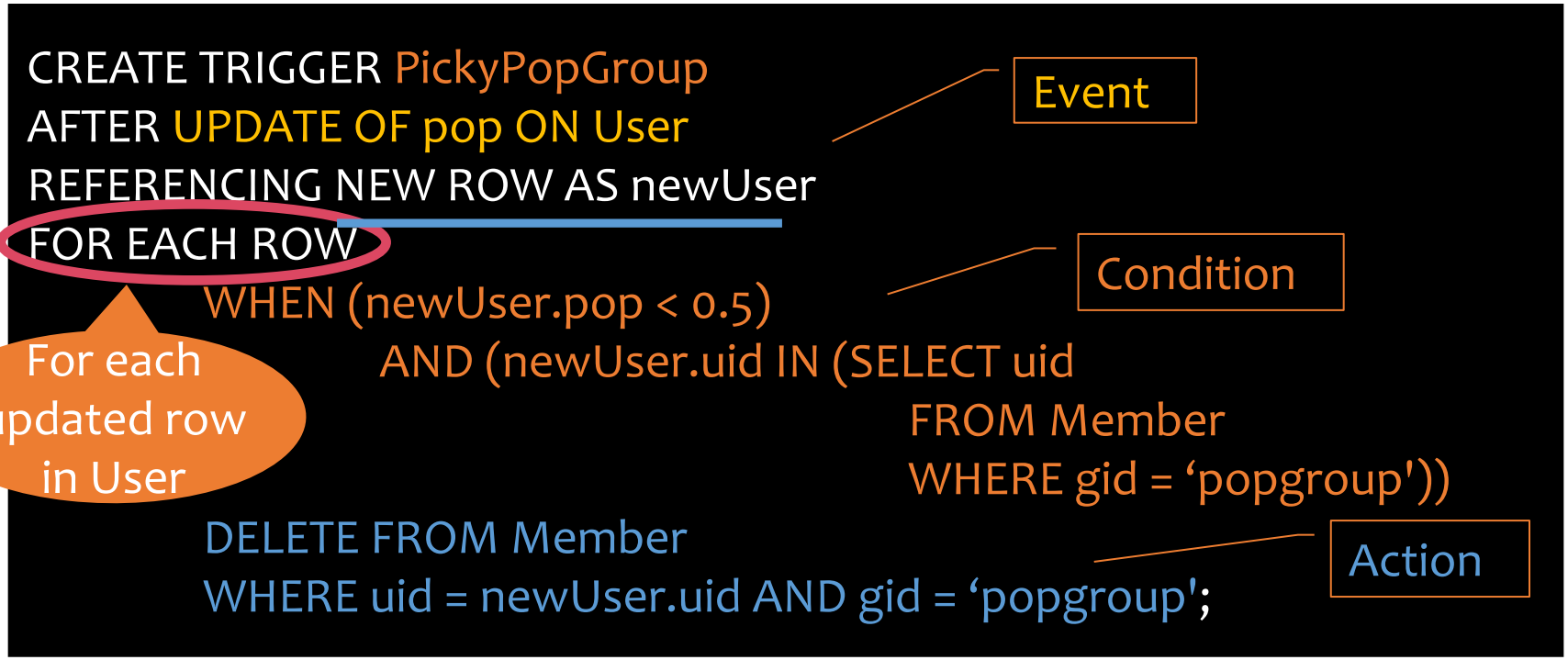
Trigger option 2 – timing

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



Trigger option 3 – granularity

- Granularity – the trigger can be activated:
  - FOR EACH ROW: execute its action for each individual affected rows in the event



Trigger option 3 – granularity

- Granularity -- the trigger can be activated:
  - FOR EACH STATEMENT: execute its action once for the entire SQL statement (rather than for each individual row affected by that statement)

```
CREATE TRIGGER PickyPopGroupAgain
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

Can only be used with **AFTER** triggers

INSTEAD OF triggers for views

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

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

0.5

0.4

For each row to be updated in AveragePop

- What does this trigger do?

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

User

…	pop	…
	0.4 +0.1	
	0.4 +0.1	
	0.5 +0.1	
	0.3 +0.1	

Transition variables/tables

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

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

AFTER Trigger

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

BEFORE Trigger

In class exercises

- If a user with *pop>0.5* is added to the User table, they must automatically belong to the '*popgroup*'. Create a trigger using FOR EACH ROW 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')
```

Event

Transition variable

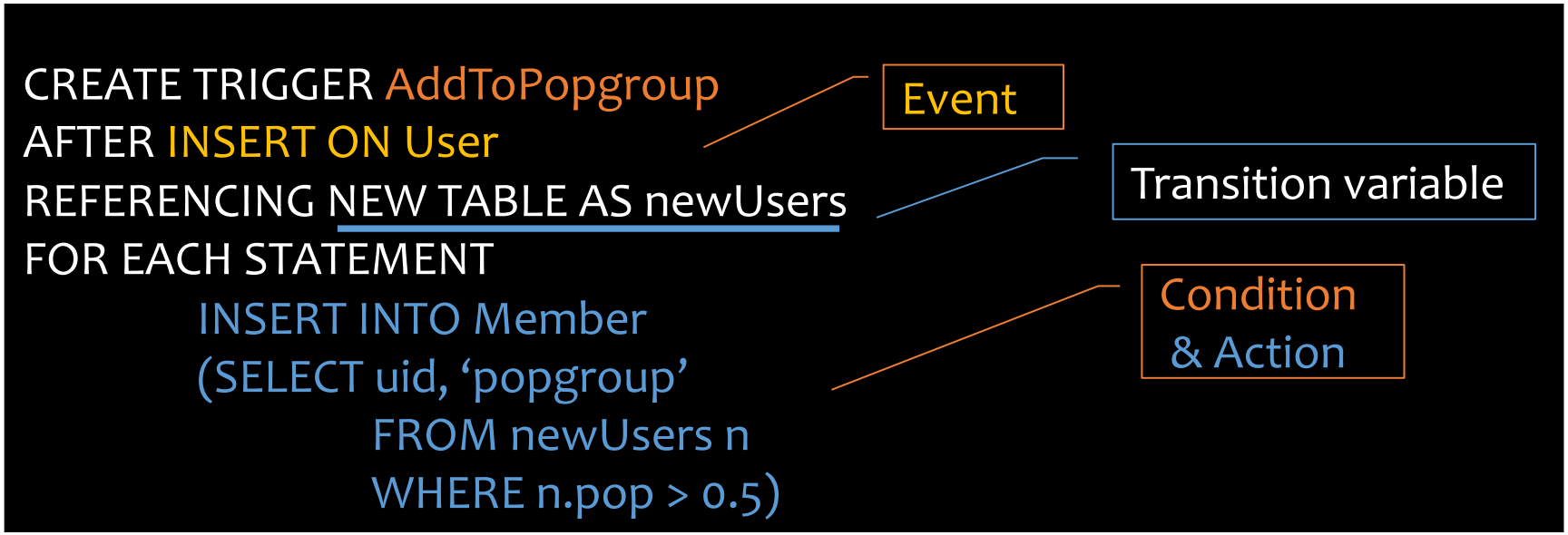
Condition

Action

In class exercises

- If a user with *pop>0.5* is added to the User table, they must automatically belong to the '*popgroup*'. Create a trigger using FOR EACH STATEMENT 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)
```



Statement- vs. Row-level triggers

- Simple row-level triggers are easier to implement
  - Statement-level triggers: require a 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 400M rows, one at a time, is not feasible due to resource constraints.
- Certain triggers only possible at the statement level

Certain triggers are only possible at statement level

```
CREATE TRIGGER MaintainAvgPop
AFTER UPDATE OF pop ON User
REFERENCING NEW TABLE AS newUsers,
                    OLD TBALE AS oldUsers
FOR EACH STATEMENT
        WHEN (0.5 > (SELECT AVG(pop) FROM User))
                    AND (100 < (SELECT COUNT(*) FROM newUsers))
        BEGIN
                    DELETE FROM User WHERE uid IN
                                (SELECT uid FROM newUsers)
                    INSERT INTO User (SELECT * FROM oldUsers)
        END
```

Event

Transition tables

Condition

Action

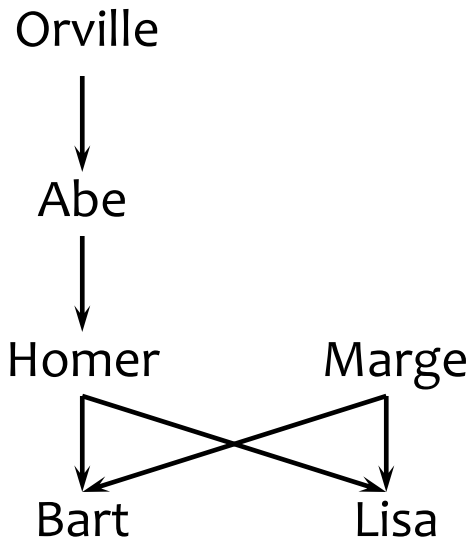
SQL features covered so far

- Basic topics:
  - Data-definition language (DDL)
  - Data-manipulation language (DML)
  - Integrity constraint
- Advanced topics:
  - View
  - Triggers (System issues)
    - Recursive firing of triggers
    - Interaction with constraints (tricky to get right!)
    - Best to avoid when alternatives exist
  - Recursion

A motivating example

Parent (*parent*, *child*)

parent	*child*
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Orville	Abe



- Example: find Bart's ancestors
- "Ancestor" has a recursive definition
  - X is Y's ancestor if
    - X is Y's parent, or
    - X is Z′s ancestor and Z is Y's ancestor

Recursion in SQL

- SQL2 had no recursion
  - You can find Bart's parents, grandparents, great grandparents, etc.

```
SELECT p1.parent AS grandparent
FROM Parent p1, Parent p2
WHERE p1.child = p2.parent AND p2.child = 'Bart';
```

  - But you cannot find all his ancestors with a single query
- SQL3 introduced recursion
  - WITH RECURSIVE clause
  - Many systems support recursion but limited functionality

Example of Ancestor Query

```
WITH RECURSIVE
Ancestor(anc, desc) AS

((SELECT parent, child FROM Parent)

UNION

(SELECT a1.anc, a2.desc
 FROM Ancestor a1, Ancestor a2
 WHERE a1.desc = a2.anc))

SELECT anc
FROM Ancestor
WHERE desc = 'Bart';
```

base case

a1.anc (X) → a1.desc(Z)
a2.anc (Z) → a2.desc (Y)

recursion step

Define a relation recursively

Query using the relation defined in WITH clause

Finding ancestors

```
WITH RECURSIVE
Ancestor(anc, desc) AS
((SELECT parent, child FROM Parent)
 UNION
 (SELECT a1.anc, a2.desc
  FROM Ancestor a1, Ancestor a2
  WHERE a1.desc = a2.anc))
```

base case

recursive step

Parent

parent	*child*
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Orville	Abe

Ancestor

anc	*desc*

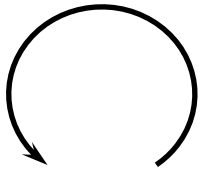
→

anc	*desc*
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Orville	Abe

→

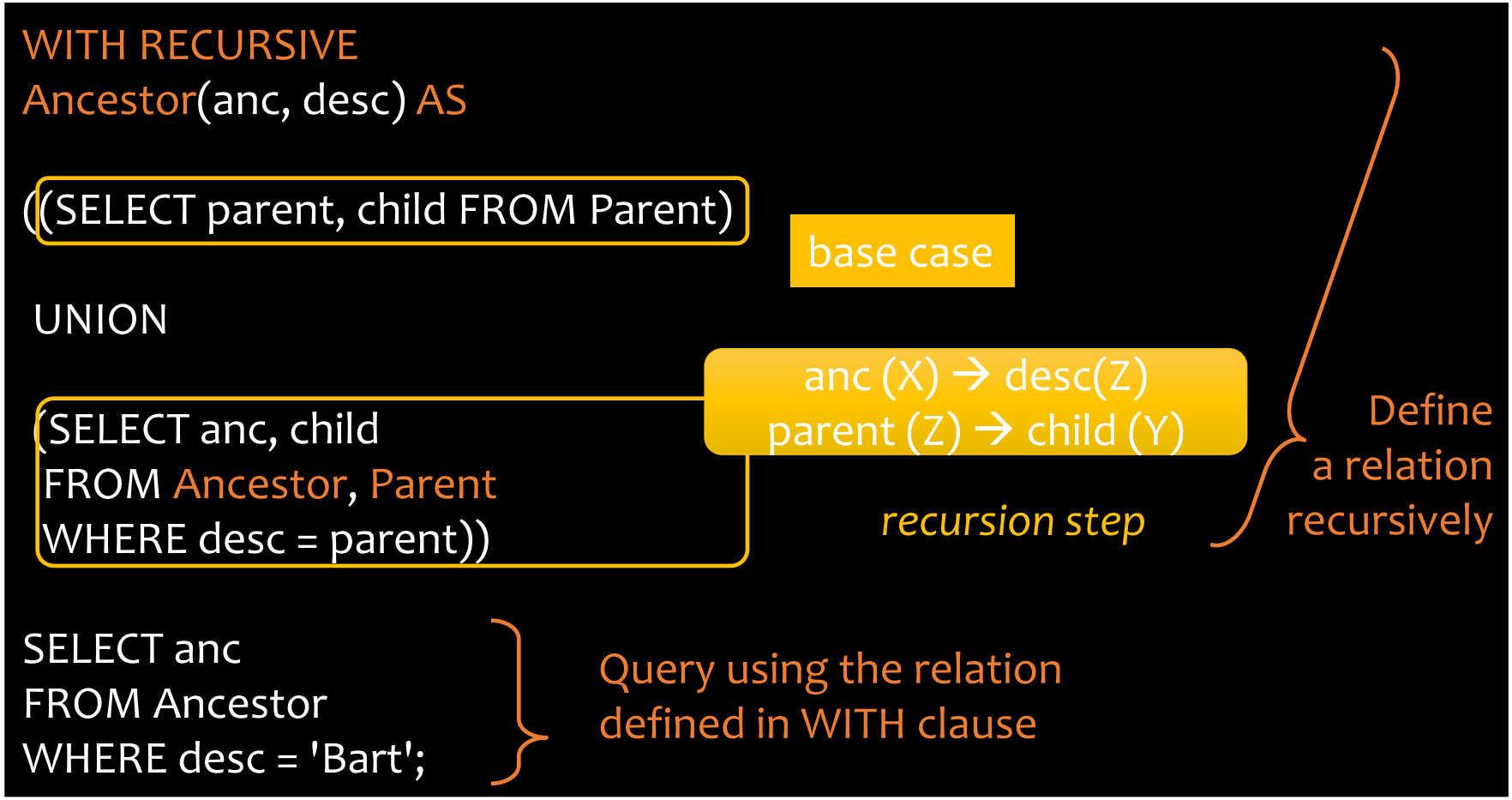
anc	*desc*
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Orville	Abe
Abe	Bart
Abe	Lisa
Orville	Homer

→



anc	*desc*
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Orville	Abe
Abe	Bart
Abe	Lisa
Orville	Homer
Orville	Bart
Orville	Lisa

Another example of Ancestor Query



Fixed point of a function

- If $f: D \rightarrow D$ is a function from a type D to itself, a fixed point of f is a value x such that $f(x) = x$
  - Example: what is the fixed point of f(x) = x/2?
  - Answer: 0 since f(0)=0

- To compute a fixed point of f:
  - Start with a "seed": $x \leftarrow x_0$
  - Compute $f(x)$
    - If $f(x) = x$, stop; x is fixed point of f
      - (Similar to base case in recursive programming)
    - Otherwise, $x \leftarrow f(x)$; repeat

Fixed point of a query

- A query q is a function that maps an input table to an output table
- A fixed point of q is a table T such that $q(T) = T$

- To compute the fixed point of q:
  - Start with executing q on a base table T_0: $T \leftarrow q(T_0)$
  - Evaluate q over T
    - If $q(T) = T$, stop; T is a fixed point of q
    - Otherwise, let $q(T)$ be the new input table; repeat

Restrictions on recursive queries

- A recursive query q must be monotonic
  - If more tuples are added to the recursive relation, q must return at least the same set of tuples as before, and possibly return additional tuples

Lecture 3

- The following is not allowed in q:
  - Aggregation on the recursive relation
  - NOT EXISTS/NOT IN in generating the recursive relation
  - Set difference (EXCEPT) whose right-hand side uses the recursive relation

SQL features covered so far

- Basic topics:
  - Data-definition language (DDL)
  - Data-manipulation language (DML)
  - Integrity constraint
- Advanced topics:
  - View
  - Triggers
  - Recursion
  - Index

Motivation of using indexes

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

- Can we go "directly" to *User* 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';
```

- For each *Member* row, can we go "directly" to *Member* rows with gid = 'popgroup'? Index on *Member.gid*

- For each *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 INDEX unique_name ON USER(name);
```

```
DROP INDEX unique_name;
```

- 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
  - $A = x$
  - $A > x$ (depends on the index)
  - Can be extended to multi-attribute index $R(A_1, A_2, \ldots, A_n)$

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

Summary of SQL

- Basic topics
  - Data-definition language (DDL): define/modify schemas, drop relations
  - Data-manipulation language (DML): query data
    - SELECT-FROM-WHERE
    - DISTINCT, UNION/EXCEPT/INTERSECT (ALL)
    - Table, Scalar, IN, EXISTS, ALL, ANY)
    - GROUP BY, HAVING
    - ORDER
    - NULL and JOIN

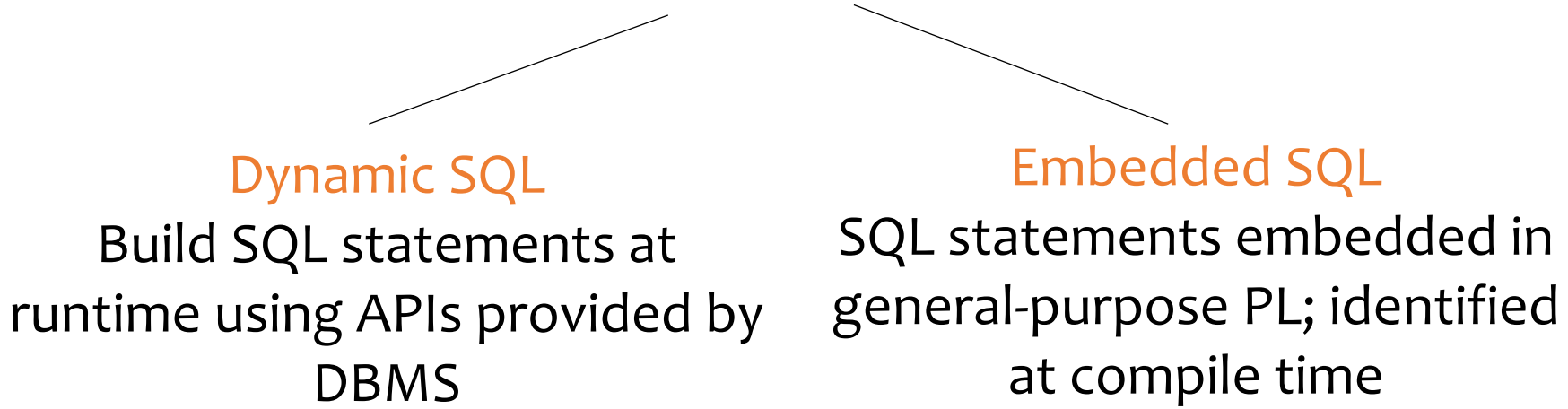
    and modify data (INSERT/DELETE/UPDATE)
  - Constraints (NOT NULL, UNIQUE, PRIMARY/FOREIGN KEY, CHECK, ASSERTION)
- Advanced topics
  - View, Triggers, Recursion, Index, Programming (optional)

SQL Programming (optional)

- Pros and cons of SQL
  - Very high-level, possible to optimize
  - Not intended for general-purpose computation

- Can SQL and general-purpose programming languages (PL) interact with each other?

YES!!



Mismatch b/w SQL and PLs (optional)

- SQL operates on a set of records at a time
- Typical low-level general-purpose programming languages operate on one record at a time

☞Solution: cursor

- Open (a result table), Get next, Close

☞Found in virtually every database language/API

- With slightly different syntaxes

Dynamic SQL

Working with SQL through an API

- Example: Python psycopg2, JDBC, ODBC (C/C++)
  - All based on the SQL/CLI (Call-Level Interface) standard

- The application program sends SQL commands to the DBMS at runtime

- Responses/results are converted to objects in the application program

Example API: Python psycopg2

```
import psycopg2
conn = psycopg2.connect(host="db.uwaterloo.ca", port=5432,
dbname="membership", user='u1', password='passwd1'))
cur = conn.cursor()
……
```

Connect to the database

An object used to query database and get results

Example API: Python psycopg2

```
import psycopg2
conn = psycopg2.connect(host="db.uwaterloo.ca", port=5432,
dbname="membership", user='u1', password='passwd1')
cur = conn.cursor()
# list all groups:
cur.execute('SELECT * FROM Group')
for gid, name in cur:
    print('Group ' + gid + ' has name ' + name)
conn.commit()
cur.close()
conn.close()
```

You can iterate over cur one tuple at a time

Commit the changes, if any, and close the cursor and the DB connection

Example API: Python psycopg2

```
# “commit” each change immediately—need to set this option just once at the start of the session
conn.set_session(autocommit=True)
# …
uid = input('Enter the user id to update: ').strip()
name = input('Enter the name to update: ').strip()
pop = float(input('Enter new pop: '))
cur.execute(“
            UPDATE User
            SET pop = %s
            WHERE uid = %s AND name = %s”, (pop, uid, name))
```

Perform parsing, semantic analysis, optimization, compilation, and final execution

What's next?

- Lecture 9: Database Design