(More) SQL

Introduction to Databases

CompSci 316 Fall 2020

DUKE
COMPUTER SCIENCE

Announcements (Thu. Sept 10)

- HW3 + Gradiance 2 posted (ER diagram)
  - Due dates: Wed September 16 11:59 pm

Recap: Basic SQL from Lecture 1-2

- Find addresses of all bars that 'Dan' frequents

```
SELECT B.address
FROM Bar B, Frequents F
WHERE B.name = F.bar
  AND F.drinker = 'Dan'
```

Bar

name	address
The Edge	108 Morris Street
Satisfaction	905 W. Main Street

We discussed

- SELECT-FROM-WHERE
- DISTINCT
- ORDER BY
- Bag vs. Set semantics (why bag?)
- Semantic of SQL evaluation (?)

drinker	bar	times_a_week
Ben	Satisfaction	2
Dan	The Edge	1
Dan	Satisfaction	2

Frequents

SQL set and bag operations

- UNION, EXCEPT, INTERSECT
  - Set semantics
    - Duplicates in input tables, if any, are first eliminated
    - Duplicates in result are also eliminated (for UNION)
  - Exactly like set ∪, −, and ∩ in relational algebra
- UNION ALL, EXCEPT ALL, INTERSECT ALL
  - Bag semantics
  - Think of each row as having an implicit count (the number of times it appears in the table)
  - Bag union: sum up the counts from two tables
  - Bag difference: proper-subtract the two counts
  - Bag intersection: take the minimum of the two counts

Examples of bag operations

Bag1

fruit
apple
apple
orange

Bag2

fruit
apple
orange
orange

```
(SELECT * FROM Bag1)
UNION ALL
(SELECT * FROM Bag2);
```

fruit
apple
apple
orange
apple
orange
orange

```
(SELECT * FROM Bag1)
EXCEPT ALL
(SELECT * FROM Bag2);
```

fruit
apple

```
(SELECT * FROM Bag1)
INTERSECT ALL
(SELECT * FROM Bag2);
```

fruit
apple
orange

Examples of set versus bag operations

Poke (*uid1*, *uid2*, *timestamp*)

- (SELECT uid1 FROM Poke)
  EXCEPT
  (SELECT uid2 FROM Poke);

- (SELECT uid1 FROM Poke)
  EXCEPT ALL
  (SELECT uid2 FROM Poke);

☞Next: how to “nest” SQL queries and write sub-queries?

Table subqueries

Poke (uid1, uid2, timestamp)

- Use query result as a table
  - In set and bag operations, FROM clauses, etc.
  - A way to "nest" queries

- Example: names of users who poked others more than others poked them

```
SELECT DISTINCT name
FROM User,
    ((SELECT uid1 AS uid FROM Poke)
     EXCEPT ALL
     (SELECT uid2 AS uid FROM Poke))
    AS T
WHERE User.uid = T.uid;
```

IN subqueries

User(uid, name, age, pop)

- x IN $(subquery)$ checks if x is in the result of $subquery$
- Example: users (all columns) at the same age as (some) Bart

Let's first try without sub-queries

```
SELECT *
FROM User
WHERE age IN (SELECT age
              FROM User
              WHERE name = 'Bart');
```

You can use NOT IN too

EXISTS subqueries

User(uid, name, age, pop)

- EXISTS (*subquery*) checks if the result of *subquery* is non-empty
- Example: users at the same age as (some) Bart
  - SELECT *
    FROM User AS u
    WHERE EXISTS (SELECT * FROM User
    WHERE name = 'Bart'
    AND age = u.age);
  - This happens to be a correlated subquery—a subquery that references tuple variables in surrounding queries

You can use NOT EXISTS too

  - How about the previous one with "IN"?

User(uid, name, age, pop)

Semantics of subqueries

Remember SQL evaluation!
FROM-WHERE-SELECT

- SELECT *
  FROM User AS u
  WHERE EXISTS (SELECT * FROM User
  WHERE name = 'Bart'
  AND age = u.age);

- For each row u in User
  - Evaluate the subquery with the value of u.age
  - If the result of the subquery is not empty, output u.*
- The DBMS query optimizer may choose to process the query in an equivalent, but more efficient way (example?)

"WITH" clause – very useful!

- You will find "WITH" clause very useful!

```
WITH Temp1 AS
        (SELECT ..... ..),
        Temp2 AS
        (SELECT ..... ..)
SELECT X, Y
FROM TEMP1, TEMP2
WHERE....
```

- Can simplify complex nested queries

Example: users at the same age as (some) Bart

```
WITH BartAge AS
  (SELECT age
   FROM User
   WHERE name = 'Bart')
SELECT U.uid, U.name, U.age, U.pop
FROM User U, BartAge B
WHERE U.age = B.age
```

WITH clause not really needed for this query!

Scalar subqueries

- A query that returns a single row can be used as a value in WHERE, SELECT, etc.
- Example: users at the same age as Bart
  - SELECT *
    FROM User
    WHERE age = (SELECT age
    FROM User
    WHERE name = 'Bart');

    What's Bart's age?
  - Runtime error if subquery returns more than one row
    - Under what condition will this error never occur?
  - What if the subquery returns no rows?
    - The answer is treated as a special value NULL, and the comparison with NULL will fail (later)

Scoping rule of subqueries

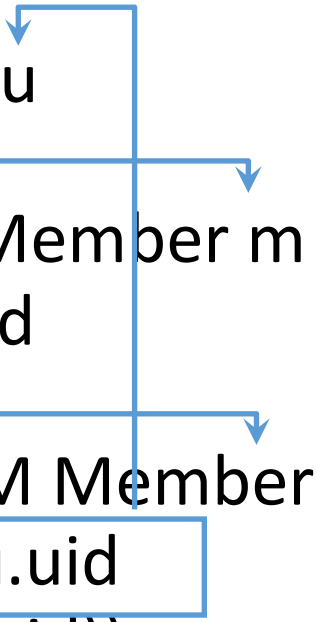
- To find out which table a column belongs to
  - Start with the immediately surrounding query
  - If not found, look in the one surrounding that; repeat if necessary

- Use *table_name.column_name* notation and AS (renaming) to avoid confusion

Another example

User(uid, name, pop)
Member(uid, gid)
Group(gid, name)

- SELECT * FROM User u
  WHERE EXISTS
  (SELECT * FROM Member m
  WHERE uid = u.uid
  AND EXISTS
  (SELECT * FROM Member
  WHERE uid = u.uid
  AND gid <> m.gid));



- What does this query return?

Quantified subqueries

Read this slide yourself
Example in class (next slide)

- A quantified subquery can be used syntactically as a value in a WHERE condition
- Universal quantification (for all):
  ... WHERE $x\ op$ ALL$(subquery)$ ...
  - True iff for all t in the result of $subquery$, $x\ op\ t$
- Existential quantification (exists):
  ... WHERE $x\ op$ ANY$(subquery)$ ...
  - True iff there exists some t in $subquery$ result such that $x\ op\ t$

  ☞ Beware
  - In common parlance, "any" and "all" seem to be synonyms
  - In SQL, ANY really means "some"

Examples of quantified subqueries

User(uid, name, pop)
Member(uid, gid)
Group(gid, name)

- Which users are the most popular?

  - SELECT *
    FROM User
    WHERE pop >= ALL(SELECT pop FROM User);

  - SELECT *
    FROM User
    WHERE NOT
      (pop < ANY(SELECT pop FROM User);

  ☞Use NOT to negate a condition

More ways to get the most popular

User(uid, name, pop)
Member(uid, gid)
Group(gid, name)

- Which users are the most popular?

  - SELECT *
    FROM User AS u
    WHERE NOT EXISTS
      (SELECT * FROM User
       WHERE pop > u.pop);

  - SELECT * FROM User
    WHERE uid NOT IN
      (SELECT u1.uid
       FROM User AS u1, User AS u2
       WHERE u1.pop < u2.pop);

☞Next: aggregates, group-by, having!

User(uid, name, age, pop)

Aggregates

- Standard SQL aggregate functions: COUNT, SUM, AVG, MIN, MAX

- Example: number of users under 18, and their average popularity
  - SELECT COUNT(*), AVG(pop)
    FROM User
    WHERE age < 18;
  - COUNT(*) counts the number of rows

Member(uid, gid)

Aggregates with DISTINCT

- Example: How many users are in some group?

  - SELECT COUNT(DISTINCT uid)
    FROM Member;

  is equivalent to:

  - SELECT COUNT(*)
    FROM (SELECT DISTINCT uid FROM Member);

Grouping

User(uid, name, age, pop)

- SELECT … FROM … WHERE …
  GROUP BY *list_of_columns*;

- Example: compute average popularity for each age group
  - SELECT age, AVG(pop)
    FROM User
    GROUP BY age;

Semantics of GROUP BY

See example
On the next slide first

SELECT … FROM … WHERE … GROUP BY …;

- Compute FROM ($\times$)
- Compute WHERE (σ)
- Compute GROUP BY: group rows according to the values of GROUP BY columns
- Compute SELECT for each group (π)
  - For aggregation functions with DISTINCT inputs, first eliminate duplicates within the group

☞ Number of groups =
number of rows in the final output

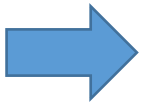
User(uid, name, age, pop)

Example of computing GROUP BY

SELECT age, AVG(pop) FROM User GROUP BY age;

uid	*name*	*age*	*pop*
142	Bart	10	0.9
857	Lisa	8	0.7
123	Milhouse	10	0.2
456	Ralph	8	0.3

Compute GROUP BY: group rows according to the values of GROUP BY columns



uid	*name*	*age*	*pop*
142	Bart	10	0.9
123	Milhouse	10	0.2
857	Lisa	8	0.7
456	Ralph	8	0.3

Compute SELECT for each group

age	*avg_pop*
10	0.55
8	0.50

User(uid, name, age, pop)

Aggregates with no GROUP BY

- An aggregate query with no GROUP BY clause = all rows go into one group

  SELECT AVG(pop) FROM User;

Group all rows into one group

Aggregate over the whole group

uid	*name*	*age*	*pop*
142	Bart	10	0.9
857	Lisa	8	0.7
123	Milhouse	10	0.2
456	Ralph	8	0.3

uid	*name*	*age*	*pop*
142	Bart	10	0.9
857	Lisa	8	0.7
123	Milhouse	10	0.2
456	Ralph	8	0.3

avg_pop
0.525

Restriction on SELECT

- If a query uses aggregation/group by, then every column referenced in SELECT must be either
  - Aggregated, or
  - A GROUP BY column

Why?

☞This restriction ensures that any SELECT expression produces only one value for each group

Examples on blackboard

User(uid, name, age, pop)

Examples of invalid queries

Which one is correct?

- SELECT uid, age
  FROM User GROUP BY age;

- SELECT uid, MAX(pop) FROM User;

HAVING

- Used to filter groups based on the group properties (e.g., aggregate values, GROUP BY column values)
- SELECT … FROM … WHERE … GROUP BY …
  HAVING *condition;*
  - Compute FROM ($\times$)
  - Compute WHERE (σ)
  - Compute GROUP BY: group rows according to the values of GROUP BY columns
  - Compute HAVING (another σ over the groups)
  - Compute SELECT (π) for each group that passes HAVING

User(uid, name, age, pop)

HAVING examples

- List the average popularity for each age group with more than a hundred users
  - SELECT age, AVG(pop)
    FROM User
    GROUP BY age
    HAVING COUNT(*) > 100;
  - Can be written using WHERE and table sub-queries
- Find average popularity for each age group over 10
  - SELECT age, AVG(pop)
    FROM User
    GROUP BY age
    HAVING age > 10;
  - Can be written using WHERE without table subqueries

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

Creating and dropping views

- Example: members of Jessica's Circle
  - CREATE VIEW JessicaCircle AS
    SELECT * FROM User
    WHERE uid IN (SELECT uid FROM Member
    WHERE gid = 'jes');
  - Tables used in defining a view are called "base tables"
    - *User* and *Member* above
- To drop a view
  - DROP VIEW JessicaCircle;

Why use views?

☞ Next: incomplete information – nulls, and outerjoins!

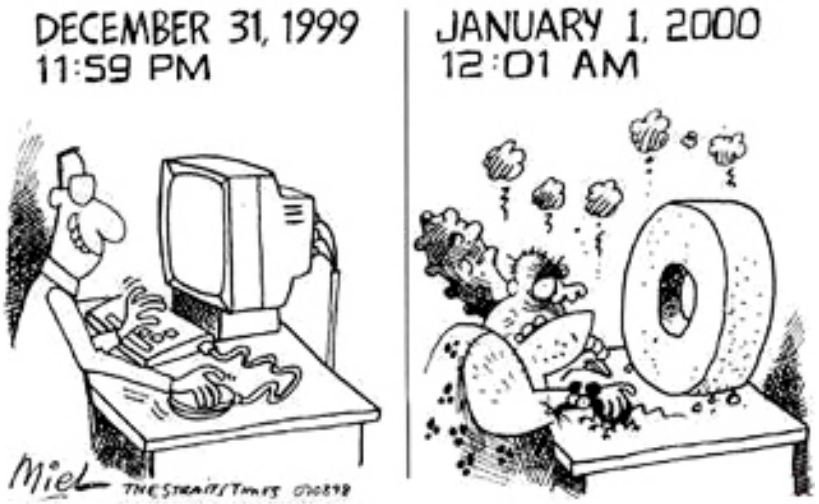
Incomplete information

- Example: *User* (*uid*, *name*, *age*, *pop*)
- Value unknown
  - We do not know Nelson's age
- Value not applicable
  - Suppose *pop* is based on interactions with others on our social networking site
  - Nelson is new to our site; what is his *pop*?

Ideas to handle unknown or missing attribute values?

Solution 1

- Dedicate a value from each domain (type)
  - *pop* cannot be −1, so use −1 as a special value to indicate a missing or invalid *pop*
  - Leads to incorrect answers if not careful
    - SELECT AVG(pop) FROM User;
  - Complicates applications
    - SELECT AVG(pop) FROM User WHERE pop <> -1;
  - Perhaps the value is not as special as you think!
    - Ever heard of the Y2K bug? "00" was used as a missing or invalid year value



http://www.90s411.com/images/y2k-cartoon.jpg

Solution 2

- A valid-bit for every column
  - *User* (*uid*,
    name, *name_is_valid*,
    age, *age_is_valid*,
    pop, *pop_is_valid*)
  - Complicates schema and queries
    - SELECT AVG(pop) FROM User
      WHERE pop_is_valid;

Solution 3

- Decompose the table; missing row = missing value
  - *UserName* (*uid*, *name*)
    UserAge (*uid*, *age*)
    UserPop (*uid*, *pop*)
  - *UserID* (*uid*)

  - Conceptually the cleanest solution
  - Still complicates schema and queries
    - How to get all information about users in a table?
    - Check yourself: Natural join doesn't work but outerjoins (soon) do -- Why?

SQL's solution

- A special value NULL
  - For every domain
  - Special rules for dealing with NULL's

- Example: *User* (*uid*, *name*, *age*, *pop*)
  - ⟨789, "Nelson", NULL, NULL⟩

Computing with NULL's

- When we operate on a NULL and another value (including another NULL) using +, −, etc., the result is NULL

- Aggregate functions ignore NULL, except COUNT(*) (since it counts rows)

Three-valued logic

- TRUE = 1, FALSE = 0, UNKNOWN = 0.5
- x AND y = $\min(x, y)$
- x OR y = $\max(x, y)$
- NOT x = $1 - x$
- When we compare a NULL with another value (including another NULL) using $=$, $>$, etc., the result is UNKNOWN
- WHERE and HAVING clauses only select rows for output if the condition evaluates to TRUE
  - UNKNOWN is not enough

Unfortunate consequences

Are these equivalent?

- SELECT AVG(pop) FROM User;
  SELECT SUM(pop)/COUNT(*) FROM User;
  - Not equivalent
  - Although AVG(pop)=SUM(pop)/COUNT(pop) still

- SELECT * FROM User;
  SELECT * FROM User WHERE pop = pop;
  - Not equivalent

☞ Be careful: NULL breaks many equivalences

Another problem

- Example: Who has NULL *pop* values?
  - SELECT * FROM User WHERE pop = NULL;
    - Does not work; never returns anything
  - SQL introduced special, built-in predicates IS NULL and IS NOT NULL
    - SELECT * FROM User WHERE pop IS NULL;
- Check yourself:
  - (SELECT * FROM User)
    EXCEPT ALL
    (SELECT * FROM User WHERE pop = pop);
    - Works, but ugly

User(uid, name, age, pop)
Member(uid, gid)

Outerjoin motivation

- Example: a master group membership list
  - SELECT g.gid, g.name AS gname,
    u.uid, u.name AS uname
    FROM Group g, Member m, User u
    WHERE g.gid = m.gid AND m.uid = u.uid;
  - What if a group is empty?
  - It may be reasonable for the master list to include empty groups as well
    - For these groups, *uid* and *uname* columns would be NULL

Outerjoin flavors and definitions

- A full outerjoin between R and S (denoted $R ⟗ S$) includes all rows in the result of $R \bowtie S$, plus
  - "Dangling" R rows (those that do not join with any S rows) padded with NULL's for S's columns
  - "Dangling" S rows (those that do not join with any R rows) padded with NULL's for R's columns
- A left outerjoin ($R ⟕ S$) includes rows in $R \bowtie S$ plus dangling R rows padded with NULL's
- A right outerjoin ($R ⟖ S$) includes rows in $R \bowtie S$ plus dangling S rows padded with NULL's

Outerjoin examples

Group

gid	*name*
abc	Book Club
gov	Student Government
dps	Dead Putting Society
nuk	United Nuclear Workers

Member

uid	*gid*
142	dps
123	gov
857	abc
857	gov
789	foo

Group ⟕ *Member*

gid	*name*	*uid*
abc	Book Club	857
gov	Student Government	123
gov	Student Government	857
dps	Dead Putting Society	142
nuk	United Nuclear Workers	NULL

Group ⟖ *Member*

gid	*name*	*uid*
abc	Book Club	857
gov	Student Government	123
gov	Student Government	857
dps	Dead Putting Society	142
foo	NULL	789

Group ⟗ *Member*

gid	*name*	*uid*
abc	Book Club	857
gov	Student Government	123
gov	Student Government	857
dps	Dead Putting Society	142
nuk	United Nuclear Workers	NULL
foo	NULL	789

Outerjoin syntax

- SELECT * FROM Group LEFT OUTER JOIN Member
  ON Group.gid = Member.gid;
  $\approx Group \overset{}{\underset{Group.gid=Member.gid}{\bowtie}} Member$
- SELECT * FROM Group RIGHT OUTER JOIN Member
  ON Group.gid = Member.gid;
  $\approx Group \overset{}{\underset{Group.gid=Member.gid}{\bowtie}} Member$
- SELECT * FROM Group FULL OUTER JOIN Member
  ON Group.gid = Member.gid;
  $\approx Group \overset{}{\underset{Group.gid=Member.gid}{\bowtie}} Member$

☞A similar construct exists for regular ("inner") joins:

- SELECT * FROM Group JOIN Member
  ON Group.gid = Member.gid;

☞These are theta joins rather than natural joins

- Return all columns in *Group* and *Member*

☞For natural joins, add keyword NATURAL; don't use ON

☞Next: how to create a table and insert/delete rows?

Creating and dropping tables

- CREATE TABLE *table_name* (..., *column_name column_type*, ...);
- DROP TABLE *table_name*;
- Examples

```
create table User(uid integer, name varchar(30),
                  age integer, pop float);
create table Group(gid char(10), name varchar(100));
create table Member(uid integer, gid char(10));
drop table Member;
drop table Group;
drop table User;
-- everything from -- to the end of line is ignored.
-- SQL is insensitive to white space.
-- SQL is insensitive to case (e.g., ...Group... is
-- equivalent to ...GROUP...).
```

INSERT

- Insert one row
  - INSERT INTO Member VALUES (789, 'dps');
    - User 789 joins Dead Putting Society
- Insert the result of a query
  - INSERT INTO Member
    (SELECT uid, 'dps' FROM User
    WHERE uid NOT IN (SELECT uid
    FROM Member
    WHERE gid = 'dps'));
    - Everybody joins Dead Putting Society!

DELETE

- Delete everything from a table
  - DELETE FROM Member;
- Delete according to a WHERE condition

Example: User 789 leaves Dead Putting Society

  - DELETE FROM Member
    WHERE uid = 789 AND gid = 'dps';

Example: Users under age 18 must be removed from United Nuclear Workers

  - DELETE FROM Member
    WHERE uid IN (SELECT uid FROM User
    WHERE age < 18)
    AND gid = 'nuk';

UPDATE

- Example: User 142 changes name to "Barney"
  - UPDATE User
    SET name = 'Barney'
    WHERE uid = 142;

- Example: We are all popular!
  - UPDATE User
    SET pop = (SELECT AVG(pop) FROM User);
    - But won't update of every row causes average *pop* to change?

    ☞Subquery is always computed over the old table

☞Next: constraints and triggers!

Constraints

- Restrictions on allowable data in a database
  - In addition to the simple structure and type restrictions imposed by the table definitions
  - Declared as part of the schema
  - Enforced by the DBMS

- Why use constraints?
  - Protect data integrity (catch errors)
  - Tell the DBMS about the data (so it can optimize better)

Types of SQL constraints

- NOT NULL
- Key
- Referential integrity (foreign key)
- Tuple- and attribute-based CHECK's
- (not covered for now -- General assertion)

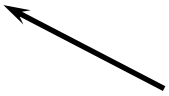
NOT NULL constraint examples

- CREATE TABLE User
  (uid INTEGER NOT NULL,
  name VARCHAR(30) NOT NULL,
  twitterid VARCHAR(15) NOT NULL,
  age INTEGER,
  pop FLOAT);
- CREATE TABLE Group
  (gid CHAR(10) NOT NULL,
  name VARCHAR(100) NOT NULL);
- CREATE TABLE Member
  (uid INTEGER NOT NULL,
  gid CHAR(10) NOT NULL);

Key declaration examples

- CREATE TABLE User
  (uid INTEGER NOT NULL PRIMARY KEY,
  name VARCHAR(30) NOT NULL,
  twitterid VARCHAR(15) NOT NULL UNIQUE,
  age INTEGER,
  pop FLOAT);
- CREATE TABLE Group
  (gid CHAR(10) NOT NULL PRIMARY KEY,
  name VARCHAR(100) NOT NULL);
- CREATE TABLE Member
  (uid INTEGER NOT NULL,
  gid CHAR(10) NOT NULL,
  PRIMARY KEY(uid, gid));

At most one primary key
Any number of unique

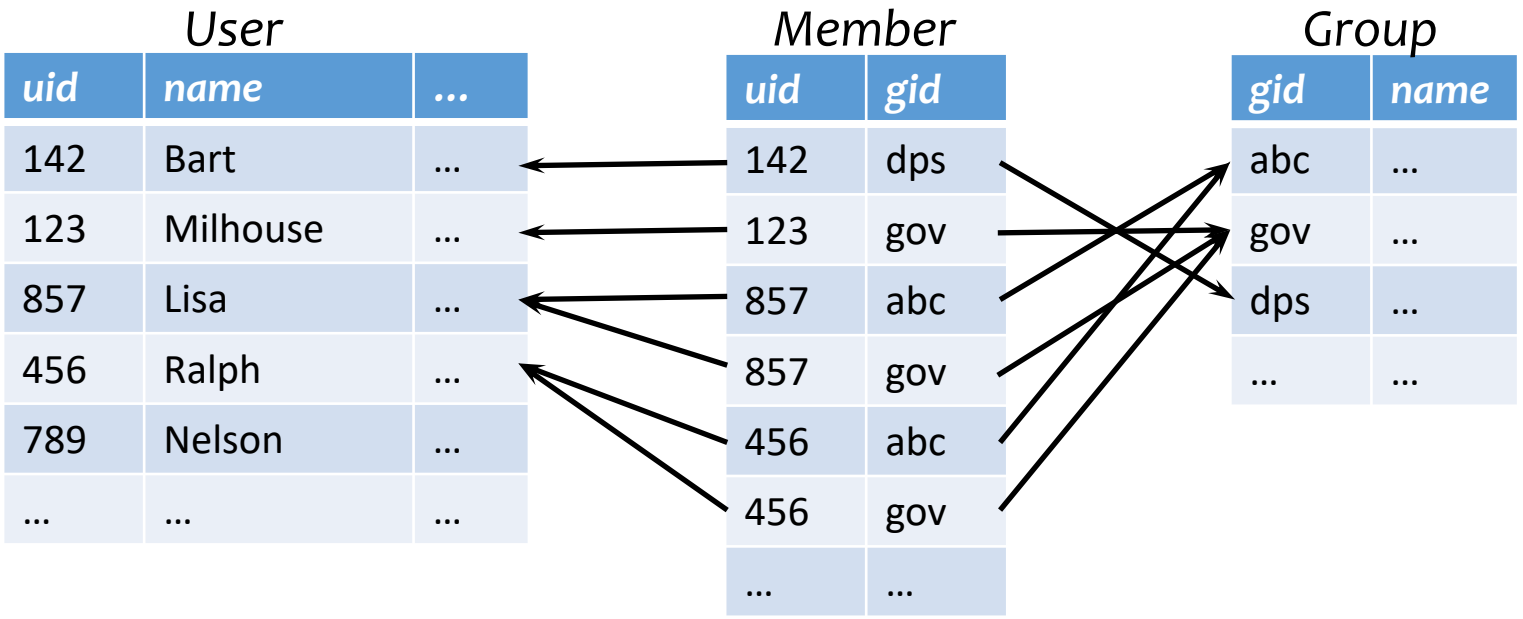


This form is required for multi-attribute keys

Referential integrity example

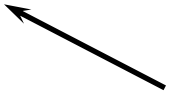
- *Member.uid* references *User.uid*
  - If an *uid* appears in *Member*, it must appear in *User*
- *Member.gid* references *Group.gid*
  - If a *gid* appears in *Member*, it must appear in *Group*

☞That is, no "dangling pointers"



Referential integrity in SQL

- Referenced column(s) must be PRIMARY KEY
- Referencing column(s) form a FOREIGN KEY
- Example
  - CREATE TABLE Member
    (uid INTEGER NOT NULL
      REFERENCES User(uid),
    gid CHAR(10) NOT NULL,
    PRIMARY KEY(uid, gid),
    FOREIGN KEY (gid) REFERENCES Group(gid));



This form is useful for multi-attribute foreign keys

Enforcing referential integrity

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

- Insert or update a *Member* row so it refers to a non-existent *uid*?
  - Reject
- Delete or update a *User* row whose *uid* is referenced by some *Member* row?
  - Reject
  - Cascade: ripple changes to all referring rows
  - Set NULL: set all references to NULL
  - All three options can be specified in SQL

Tuple- and attribute-based CHECK's

- Associated with a single table
- Only checked when a tuple/attribute is inserted/updated
  - Reject if condition evaluates to FALSE
  - TRUE and UNKNOWN are fine
    - (unlike only TRUE in WHERE conditions!)
- Examples:
  - CREATE TABLE User(...
    age INTEGER CHECK(age IS NULL OR age > 0),
    ...);
  - CREATE TABLE Member
    (uid INTEGER NOT NULL,
    CHECK(uid IN (SELECT uid FROM User)),
    ...);

Is it a referential integrity constraint?
Not quite; not checked when *User* is modified

"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. Examples?
  - 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

- Example:
  - Event: some user's popularity is updated
  - Condition: the user is a member of "Jessica's Circle," and *pop* drops below 0.5
  - Action: kick that user out of Jessica's Circle

Jessica is picky about her group members!

http://pt.simpsons.wikia.com/wiki/Arquivo:Jessica_lovejoy.jpg

Trigger example (Row Level)

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

Event

Condition

Action

Trigger options

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

- Possible events include:
  - INSERT ON *table*
  - DELETE ON *table*
  - UPDATE [OF *column*] ON *table*
- Granularity—trigger can be activated:
  - FOR EACH ROW modified
  - FOR EACH STATEMENT that performs modification
- Timing—action can be executed:
  - AFTER or BEFORE the triggering event
  - INSTEAD OF the triggering event on views (more later)

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.

- AFTER INSERT statement-level triggers
  - Can use only NEW TABLE
- AFTER UPDATE row-level triggers
  - Can use only OLD ROW and NEW ROW
- BEFORE DELETE row-level triggers
  - Can use only OLD ROW
- etc.

Statement-level trigger example

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

Event

Action

BEFORE trigger example

- Never allow age to decrease

```
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

Statement- vs. 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

SQL features covered so far

- Query
- Modification
- Views
- Constraints
- Triggers

- Still a lot more features of SQL not covered
- Learn some of them yourself as you play with SQL queries!