(More) SQL

Introduction to Databases

CompSci 316 Spring 2020

DUKE COMPUTER SCIENCE

1

Announcements (Tue. Feb. 4)

- HW3 posted (all questions now)
  - Due dates: Q1-Q3: Tuesday Feb 11 11:59 pm
  - Q4-Q5: Thursday Feb 13 11:59 pm
  - Many parts, keep working on it!
- Please form your groups by this Thursday Feb 6
  - So that we can help you find a group if needed well before MS1 is due
  - Project formation spreadsheet shared
  - 5 members for standard projects please! (otherwise we may have to shuffling later, better if you do it yourself)
  - If you want to do an open project, let me know asap

2

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

3

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

4

Examples of bag operations

Bag1

fruit
apple
apple
orange

Bag2

fruit
apple
orange
orange

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

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

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

5

Examples of set versus bag operations

Poke (uid1, uid2, timestamp)

What do these queries return?

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

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

6

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

7

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;
```

8

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

You can use NOT IN too

9

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
  - This happens to be a correlated subquery—a subquery that references tuple variables in surrounding queries
  - How about the previous one with "IN"?

You can use NOT EXISTS too

10

Semantics of subqueries

User(uid, name, age, pop)

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

Remember SQL evaluation!
FROM-WHERE-SELECT

- 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?)

11

"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!

12

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)

13

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

14

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?

15

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"

16

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

17

More ways to get the most popular

Practice queries – check yourself

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);

18

☞Next: aggregates, group-by, having!

19

Aggregates

User(uid, name, age, pop)

- 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

20

Aggregates with DISTINCT

Member(uid, gid)

- 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);

21

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;

22

Semantics of GROUP BY

See example
On the next slide first

SELECT … FROM … WHERE … GROUP BY …;

- Compute FROM (×)
- 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

23

Example of computing GROUP BY

User(uid, name, age, pop)

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

24

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

25

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?

Examples on blackboard

26

Examples of invalid queries

Which one is correct?

- SELECT uid, age
  FROM User GROUP BY age;
- SELECT uid, MAX(pop) FROM User;

27

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

28

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

29

☞Next: incomplete information and nulls!

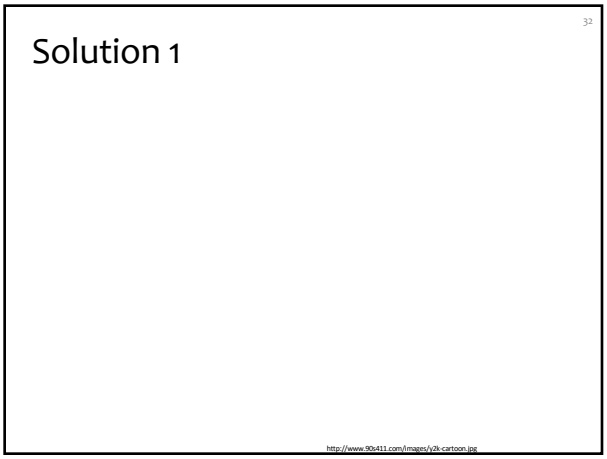
30

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*?

31

Solution 1



32

Solution 2

33

Solution 3

34

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⟩

35

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)

36

37

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

37

38

Unfortunate consequences

- 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

38

39

Another problem

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

39

40

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

40

41

Outerjoin flavors and definitions

- A full outerjoin between R and S (denoted $R \bowtie\!\!\!\!\!\!\circ\, S$) 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 ($R \bowtie\!\!\!\!\!\!\circ\, S$) includes rows in $R \bowtie S$ plus dangling R rows padded with NULL's
- A right outerjoin ($R \bowtie\!\!\!\!\!\!\circ\, S$) includes rows in $R \bowtie S$ plus dangling S rows padded with NULL's

41

42

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

42

43

Outerjoin syntax

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

☞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

43

44

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

44

45

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...).
```

45

46

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!

46

47

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';

47

48

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

48

49

☞Next: constraints!

49

50

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)

50

51

Types of SQL constraints

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

51

52

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);

52

53

Key declaration

- At most one PRIMARY KEY per table
  - Typically implies a primary index
  - Rows are stored inside the index, typically sorted by the primary key value ⇒ best speedup for queries
- Any number of UNIQUE keys per table
  - Typically implies a secondary index
  - Pointers to rows are stored inside the index ⇒ less speedup for queries

53

54

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));

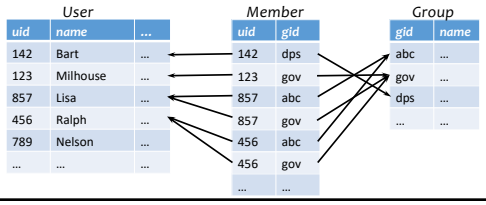
This form is required for multi-attribute keys

54

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"



55

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

56

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

57

Deferred constraint checking

- No-chicken-no-egg problem
  - CREATE TABLE Dept
    (name CHAR(20) NOT NULL PRIMARY KEY,
    chair CHAR(30) NOT NULL
    REFERENCES Prof(name));
    CREATE TABLE Prof
    (name CHAR(30) NOT NULL PRIMARY KEY,
    dept CHAR(20) NOT NULL
    REFERENCES Dept(name));
  - The first INSERT will always violate a constraint!
- Deferred constraint checking is necessary
  - Check only at the end of a transaction
  - Allowed in SQL as an option
- Curious how the schema was created in the first place?
  - ALTER TABLE ADD CONSTRAINT (read the manual!)

58

General assertion

- CREATE ASSERTION *assertion_name*
  CHECK *assertion_condition*;
- *assertion_condition* is checked for each modification that could potentially violate it
- Example: *Member.uid* references *User.uid*
  - CREATE ASSERTION MemberUserRefIntegrity
    CHECK (NOT EXISTS
    (SELECT * FROM Member
    WHERE uid NOT IN
    (SELECT uid FROM User)));

☞In SQL3, but not all (perhaps no) DBMS supports it

59

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
- 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

60

61

SQL features covered so far

- Query
  - SELECT-FROM-WHERE statements
  - Set and bag operations
  - Table expressions, subqueries
  - Aggregation and grouping
  - Ordering
  - Outerjoins
- Modification
  - INSERT/DELETE/UPDATE
- Constraints

☞Next: triggers, views, indexes

61