Review Lectures 2-4

Introduction to Database Management

CS348 Fall 2022

Announcements (Thur. Sep 22)

- Project milestone 0 due by Sep 27 (Tue), 11:59pm
  - Form a team on Learn
  - Report.pdf and link to GitHub repo
  - Not graded, but very important!

- Assignment 1 due by Sep 29 (Thur), 11:59pm
  - Part 1: general questions and r.a.
    - Submit via Crowdmark
  - Part 2: writing SQL on DB2 on school servers (try soon)
    - Submit via Marmoset

Outline

- Lecture 2: Intro to the relational model
    - Relational data model
    - Relational algebra

- Lectures 3 & 4: SQL (1) & (2)

Relational data model

- A database is a collection of relations (or tables)
- Each relation has a set of attributes (or columns)
- Each attribute has a name and a domain (or type)
  - The domains are required to be atomic
- Each relation contains a set of tuples (or rows)
  - Each tuple has a value for each attribute of the relation
  - Duplicate tuples are not allowed
    - Two tuples are duplicates if they agree on all attributes

☞Simplicity is a virtue!

Example for Relational data model

User

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

tuples (or rows)

Duplicates are not allowed

Ordering of rows doesn't matter (even though output is always in some order)

Group

gid	*name*
abc	Book Club
gov	Student Government
dps	Dead Putting Society
…	…

Member

uid	*gid*
142	dps
123	gov
857	abc
857	gov
456	abc
456	gov
…	…

Schema vs. instance

- Schema (metadata)
  - Specifies the logical structure of data
  - Is defined at setup time, rarely changes

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

- Instance
  - Represents the data content
  - Changes rapidly, but always conforms to the schema

User: {⟨142, Bart, 10, 0.9⟩, ⟨857, Milhouse, 10, 0.2⟩, … }
Group: {⟨abc, Book Club⟩, ⟨gov, Student Government⟩, … }
Member: {⟨142, dps⟩, ⟨123, gov⟩, … }

Types of integrity constraints

- Tuple-level
  - Domain restrictions, attribute comparisons, etc.
    - E.g. *age* cannot be negative
- Relation-level
  - Key constraints (focus in this lecture)
    - E.g. *uid* should be unique in the *User* relation
  - Functional dependencies (Textbook, Ch. 7)
- Database-level
  - Referential integrity – foreign key (focus in this lecture)
    - *uid* in *Member* must refer to a row in *User* with the same *uid*

Key (Candidate Key)

Def: A set of attributes K for a relation R if

- Condition 1: In no instance of R will two different tuples agree on all attributes of K
  - That is, K can serve as a "tuple identifier"
- Condition 2: No proper subset of K satisfies the above condition
  - That is, K is minimal

- Example: *User* (*uid*, *name*, *age*, *pop*)
  - *uid* is a key of *User*
  - *age* is not a key (not an identifier)
  - {*uid*, *name*} is not a key (not minimal), but a superkey

Only Check Condition 1

More examples of keys

- *Member* (*uid*, *gid*)
  - {*uid*, *gid*}
  - ☞ A key can contain multiple attributes
- *Address* (*street_address*, *city*, *state*, *zip*)
  - Key 1: {*street_address*, *city*, *state*}
  - Key 2: {*street_address*, *zip*}
  - ☞ A relation can have multiple keys!

Member

uid	*gid*
142	dps
123	gov
857	abc
857	gov
456	abc
456	gov
…	…

- Primary key: a designated candidate key in the schema declaration
  - Underline all its attributes, e.g., *Address* (<u>*street_address*</u>, *city*, *state*, <u>*zip*</u>)

"Pointers" to other rows

- Foreign key: primary key of one relation appearing as attribute of another relation

User

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

Group

gid	*name*
abc	Book Club
gov	Student Government
dps	Dead Putting Society
…	…

Member

uid	*gid*
142	dps
123	gov
857	abc
857	gov
456	abc
456	gov

“Pointers” to other rows

- Referential integrity: A tuple with a non-null value for a foreign key that does not match the primary key value of a tuple in the referenced relation is not allowed.

Member

uid	gid
142	dps
123	gov
857	~~Xi~~
857	gov
456	abc
456	gov
…	…

Group

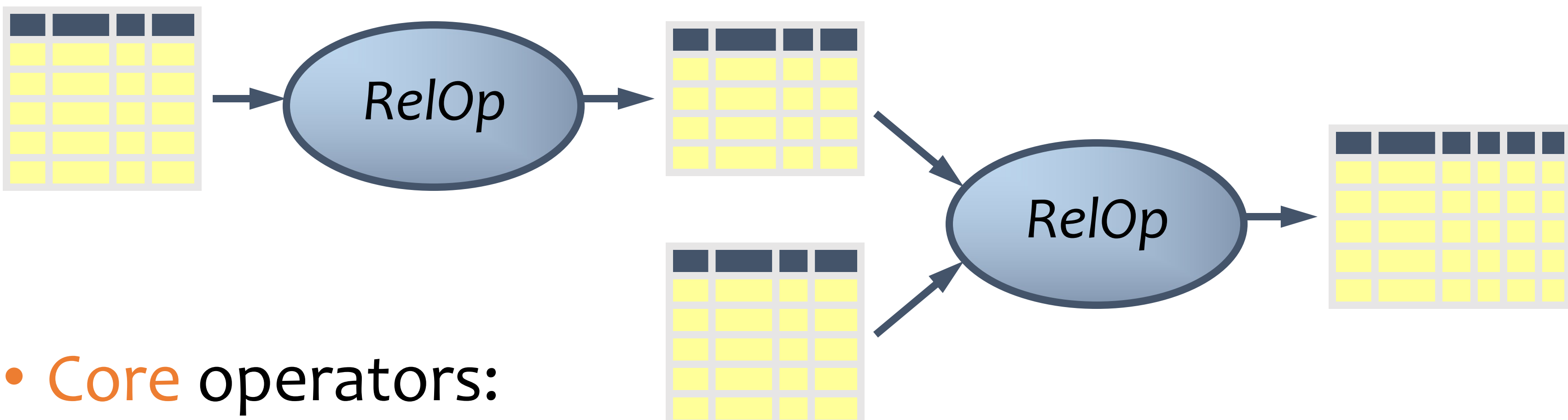
gid	name
abc	Book Club
gov	Student Government
dps	Dead Putting Society

Outline

- Lecture 2: Intro to the relational model
  - Relational data model
  - Relational algebra
- Lectures 3 & 4: SQL (1) & (2)

Relational algebra

A language for querying relational data based on "operators"



- Core operators:
  - Selection, projection, cross product, union, difference, and renaming
- Additional, derived operators:
  - Join, natural join, intersection, etc.
- Compose operators to make complex queries

Summary of operators

Core Operators

1. Selection: $\sigma_p R$
2. Projection: $\pi_L R$
3. Cross product: $R \times S$
4. Union: $R \cup S$
5. Difference: $R - S$
6. Renaming: $\rho_{S(A_1 \to A_1', A_2 \to A_2', \ldots)} R$

   Does not really add "processing" power

Derived Operators

1. Join: $R \bowtie_p S$
2. Natural join: $R \bowtie S$
3. Intersection: $R \cap S$

Note: Only use these operators for assignments & quiz

More on selection

- Selection condition can include any column of R, constants, comparison ($=$, $\leq$, etc.) and Boolean connectives ($\wedge$: and, $\vee$: or, $\neg$: not)
  - Example: users with popularity at least 0.9 and age under 10 or above 12

$$\sigma_{pop \geq 0.9 \wedge (age < 10 \vee age > 12)}\ User$$

- You must be able to evaluate the condition over each single row of the input table!
  - Example: the most popular user

$$\sigma_{pop \geq every\ pop\ in\ User}\ User$$ **WRONG!**

More on projection

- Duplicate output rows are removed (by definition)
  - Example: user ages

$$\pi_{age}\ User$$

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

age
10
[redacted]
8
[redacted]
…

Core operator 3: Cross product

- Input: two tables R and S
- Natation: $R \times S$
- Purpose: pairs rows from two tables
- Output: for each row r in R and each s in S, output a row rs (concatenation of r and s)

Derived operator 1: Join

- Info about users, plus IDs of their groups

$$User \bowtie_{User.uid=Member.uid} Member$$

uid	name	age	pop
123	Milhouse	10	0.2
857	Lisa	8	0.7
…	…	…	…

uid	gid
123	gov
857	abc
857	gov
…	…

$\bowtie_{User.uid=Member.uid}$

Prefix a column reference with table name and "." to disambiguate identically named columns from different tables

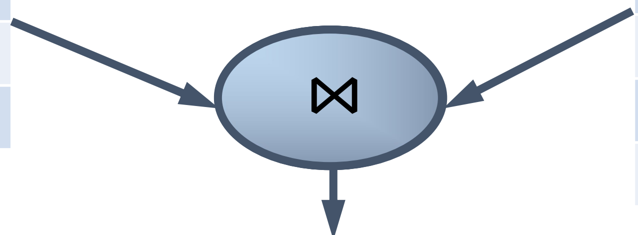
uid	name	age	pop	uid	gid
123	Milhouse	10	0.2	123	gov
[redacted]					
857	Lisa	8	0.7	857	abc
857	Lisa	8	0.7	857	gov
…	…	…	…	…	…

Derived operator 2: Natural join

$$User \bowtie Member$$
$$= \pi_{uid,name,age,pop,gid}\left(User \bowtie_{\substack{User.uid= \\ Member.uid}} Member\right)$$

uid	name	age	pop
123	Milhouse	10	0.2
857	Lisa	8	0.7
…	…	…	…

uid	gid
123	gov
857	abc
857	gov
…	…



uid	name	age	pop	[redacted]	gid
123	Milhouse	10	0.2	[redacted]	gov
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
857	Lisa	8	0.7	[redacted]	abc
857	Lisa	8	0.7	[redacted]	gov
…	…	…	…	[redacted]	…

Core operator 4: Union

- Input: two tables R and S
- Notation: $R \cup S$
  - R and S must have identical schema
- Output:
  - Has the same schema as R and S
  - Contains all rows in R and all rows in S (with duplicate rows removed)

uid	gid
123	gov
857	abc

$\cup$

uid	gid
123	gov
901	edf

=

uid	gid
123	gov
857	abc
901	edf

Core operator 5: Difference

- Input: two tables R and S
- Notation: $R - S$
  - R and S must have identical schema
- Output:
  - Has the same schema as R and S
  - Contains all rows in R that are not in S

uid	*gid*
123	gov
857	abc

$-$

uid	*gid*
123	gov
901	edf

$=$

uid	*gid*
857	abc

Derived operator 3: Intersection

- Input: two tables R and S
- Notation: $R \cap S$
  - R and S must have identical schema
- Output:
  - Has the same schema as R and S
  - Contains all rows that are in both R and S
- Shorthand for $R - (R - S)$
- Also equivalent to $S - (S - R)$
- And to $R \bowtie S$

Core operator 6: Renaming

- Input: a table R
- Notation: $\rho_S\, R$, $\rho_{(A_1 \to A_1', \ldots)} R$, or $\rho_{S(A_1 \to A_1', \ldots)} R$
- Purpose: "rename" a table and/or its columns
- Output: a table with the same rows as R, but called differently

Member

uid	*gid*
123	gov
857	abc

$$\rho_{M1(uid \to uid_1, gid \to gid_1)} Member$$

M1

uid1	*gid1*
123	gov
857	abc

9. Basic operator: Renaming

- IDs of users who belong to at least two groups

$$Member \bowtie_{?} Member$$

uid	*gid*
100	gov
100	abc
200	gov

uid	*gid*
100	gov
100	abc
200	gov

$\bowtie_{?}$

uid	*gid*	*uid*	*gid*
~~100~~	~~gov~~	~~100~~	~~gov~~
100	gov	100	abc
~~100~~	~~gov~~	~~200~~	~~gov~~
100	abc	100	gov
~~100~~	~~abc~~	~~100~~	~~abc~~
~~100~~	~~abc~~	~~200~~	~~gov~~
~~200~~	~~gov~~	~~100~~	~~gov~~
~~200~~	~~gov~~	~~100~~	~~abc~~
~~200~~	~~gov~~	~~200~~	~~gov~~

Condition 1: same uid

Condition 2: different gids

Renaming example

- IDs of users who belong to at least two groups

$$Member \bowtie_{?} Member$$

$$\pi_{uid}\left(Member \bowtie_{\substack{Member.uid=Member.uid \,\wedge \\ Member.gid \neq Member.gid}} Member\right)$$

WRONG!

$$\pi_{uid_1}\begin{pmatrix} \rho_{(uid \to uid_1, gid \to gid_1)} Member \\ \bowtie_{uid_1 = uid_2 \,\wedge\, gid_1 \neq gid_2} \\ \rho_{(uid \to uid_2, gid \to gid_2)} Member \end{pmatrix}$$

Expression tree notation

$$\pi_{uid_1}$$

$$\bowtie_{uid_1=uid_2 \wedge gid_1 \neq gid_2}$$

$$\rho_{(uid \to uid_1, gid \to gid_1)} \qquad \rho_{(uid \to uid_2, gid \to gid_2)}$$

$$Member \qquad Member$$

Take-home Exercises

- Exercise 1: IDs of groups who have at least 2 users?
- Exercise 2: IDs of users who belong to at least three groups?

User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)

A trickier example

User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)

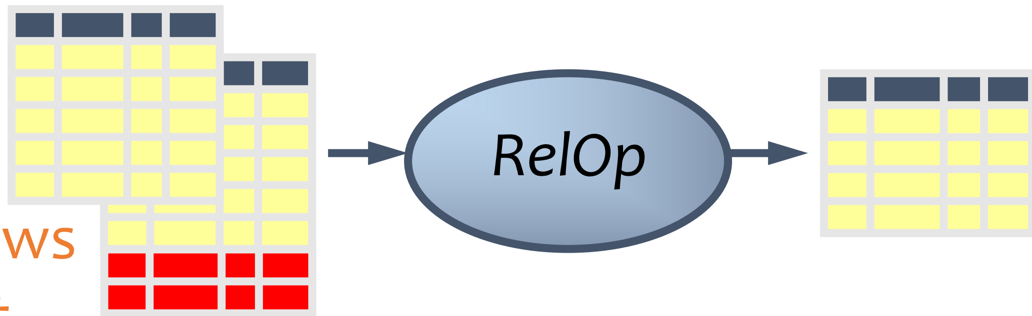
- Who are the most popular?
  - Who do NOT have the highest *pop* rating?
  - Whose *pop* is lower than somebody else's?

$$\pi_{uid} User - \pi_{User_1.uid}\left(\rho_{User_1} User \bowtie_{User_1.pop<User_2.pop} \rho_{User_2} User\right)$$

A deeper question:
When (and why) is "—" needed?

Non-monotone operators

Add more rows to the input…



What happens to the output?

- If some old output rows may become invalid, and need to be removed → the operator is non-monotone

- Example: difference operator $R - S$

R

uid	*gid*
123	gov
857	abc

−

S

uid	*gid*
123	gov
901	edf
857	**abc**

=

uid	*gid*
~~857~~	~~abc~~

This old row becomes invalid because the new row added to S

Classification of relational operators

• Selection: $\sigma_p R$	Monotone
• Projection: $\pi_L R$	Monotone
• Cross product: $R \times S$	Monotone
• Join: $R \bowtie_p S$	Monotone
• Natural join: $R \bowtie S$	Monotone
• Union: $R \cup S$	Monotone
• Difference: $R - S$	Monotone w.r.t. R; non-monotone w.r.t S
• Intersection: $R \cap S$	Monotone

Why do we need core operator X?

- Difference
  - The only non-monotone operator
- Projection
  - The only operator that removes columns
- Cross product
  - The only operator that adds columns
- Union
  - ?
- Selection
  - ?

Extensions to relational algebra

- Duplicate handling ("bag algebra")
- Grouping and aggregation
- "Extension" (or "extended projection") to allow new column values to be computed

☞ All these will come up when we talk about SQL

☞ But for now we will stick to standard relational algebra without these extensions

Outline

- Lecture 2: Intro to the relational model
  - Relational data model
  - Relational algebra

- Lectures 3 & 4: SQL (1) & (2)
  - Data-definition language (DDL): define/modify schemas, delete relations
  - Data-manipulation language (DML): query information, and insert/delete/modify tuples
  - Integrity constraints: specify constraints that the data stored in the database must satisfy

DDL

User (<u>uid</u> int, *name* string, *age* int, *pop* float)
Group (<u>gid</u> string, *name* string)
Member (<u>uid</u> int, <u>gid</u> string)

- CREATE TABLE *table_name* (…, *column_name column_type*, …);

```
CREATE TABLE User(uid DECIMAL(3,0), name VARCHAR(30), age DECIMAL
(2,0), pop DECIMAL(3,2));
CREATE TABLE Group (gid CHAR(10), name VARCHAR(100));
CREATE TABLE Member (uid DECIMAL (3,0), gid CHAR(10));
```

- DROP TABLE *table_name*;

```
DROP TABLE User;
DROP TABLE Group;
DROP TABLE Member;
```

-- everything from -- to the end of line is ignored.
-- SQL is insensitive to white space.
-- SQL is insensitive to case (e.g., ...CREATE... is
-- equivalent to ...create...).

Basic queries for DML: SFW statement

- SELECT $A_1, A_2, \ldots, A_n$
  FROM $R_1, R_2, \ldots, R_m$
  WHERE $condition$;

- Also called an SPJ (select-project-join) query

- Corresponds to (but not really equivalent to) relational algebra query:

$$\pi_{A_1, A_2, \ldots, A_n}\left(\sigma_{condition}(R_1 \times R_2 \times \cdots \times R_m)\right)$$

Forcing set semantics

- ID's of all pairs of users that belong to one group

```
SELECT m1.uid AS uid1, m2.uid AS uid2
    FROM Member AS m1, Member AS m2
    WHERE m1.gid = m2.gid
        AND m1.uid > m2.uid;
```

→Say Lisa and Ralph are in both the book club and the student government, they id pairs will appear twice

- Remove duplicate (uid1, uid2) pairs from the output

```
SELECT DISTINCT m1.uid AS uid1, m2.uid AS uid2
    FROM Member AS m1, Member AS m2
    WHERE m1.gid = m2.gid;
        AND m1.uid > m2.uid;
```

Semantics of SFW

- SELECT [DISTINCT] $E_1, E_2, \ldots, E_n$
  FROM $R_1, R_2, \ldots, R_m$
  WHERE $condition;$
- For each t_1 in R_1:
  For each t_2 in R_2: … …
  For each t_m in R_m:
  If $condition$ is true over $t_1, t_2, \ldots, t_m$:
  Compute and output $E_1, E_2, \ldots, E_n$ as a row

  If DISTINCT is present
  Eliminate duplicate rows in output
- $t_1, t_2, \ldots, t_m$ are often called tuple variables

SQL set and bag operations

- Set: UNION, EXCEPT, INTERSECT
  - Exactly like set ∪, −, and ∩ in relational algebra
  - Duplicates in input tables, if any, are first eliminated
  - Duplicates in result are also eliminated (for UNION)

Bag1

fruit
apple
apple
orange

Bag2

fruit
orange
orange
orange

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

fruit
apple
orange

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

fruit
apple

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

fruit
orange

SQL set and bag operations

- Set: UNION, EXCEPT, INTERSECT
  - Exactly like set ∪, −, and ∩ in relational algebra
- Bag: UNION ALL, EXCEPT ALL, INTERSECT ALL
  - Think of each row as having an implicit count (the number of times it appears in the table)

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

sum up the counts from two tables

Bag1

fruit
apple
apple
orange

apple: 2
orange:1

Bag2

fruit
apple
orange
orange

apple: 1
orange:2

fruit
apple
apple
orange
apple
orange
orange

apple: 3
orange:3

SQL features covered so far

- SELECT-FROM-WHERE statements
- Set and bag operations

☞Next: how to nest SQL queries

Table subqueries

- Use query result as a table
  - In set and bag operations, FROM clauses, etc.

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

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

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’);
```

- When can this query go wrong?
  - Return more than 1 row
  - Return no rows

IN subqueries

- x IN ($subquery$) checks if x is in the result of $subquery$

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

```
SELECT *
FROM User,
WHERE age IN (SELECT age
              FROM User
              WHERE name = ‘Bart’);
```

EXISTS subqueries

- 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

Quantified subqueries

- Universal quantification (for all):
  - ... WHERE x op ALL($subquery$) ...
  - True iff for all t in the result of $subquery$, x op t

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

- Existential quantification (exists):
  - ... WHERE x op ANY($subquery$) ...
  - True iff there exists some t in $subquery$ result s.t. x op t

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

More ways to get the most popular

- Which users are the most popular?

```
Q1. SELECT *
FROM User
WHERE pop >= ALL(SELECT pop FROM User);
```

```
Q2. SELECT *
FROM User
WHERE NOT
   (pop < ANY(SELECT pop FROM User);
```

EXISTS or IN?

```
Q3. SELECT *
FROM User AS u
WHERE NOT [EXITS or IN?]
   (SELECT * FROM User
    WHERE pop > u.pop);
```

```
Q4. SELECT * FROM User
WHERE uid NOT [EXISTS or IN?]
   (SELECT u1.uid
    FROM User AS u1, User AS u2
    WHERE u1.pop < u2.pop);
```

SQL features covered so far

- SELECT-FROM-WHERE statements
- Set and bag operations
- Subqueries
  - Subqueries allow queries to be written in more declarative ways (recall the "most popular" query)
  - But in many cases, they don't add expressive power

☞Next: aggregation and grouping

Aggregates

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

- Example: number of users under 18, and their average popularity
  - COUNT(*) counts the number of rows

```
SELECT COUNT(*), AVG(pop)
FROM User
WHERE age <18;
```

Aggregates with DISTINCT

- Example: How many users are in some group?

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

Is equivalent to

```
SELECT COUNT(DISTINCT uid)
FROM Member;
```

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

Aggregates with no GROUP BY

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

```
SELECT AVG(pop) FROM User;
```

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

Group all rows into one 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

Aggregate over the whole group

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

```
SELECT uid, age FROM User GROUP BY age;
```

WRONG!

```
SELECT uid, MAX(pop) FROM User;
```

WRONG!

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 subqueries

```
SELECT T.age, T.apop
FROM (SELECT age, AVG(pop) AS apop, COUNT(*) AS gsize
           FROM User GROUP BY age) AS T
WHERE T.gsize>100;
```

ORDER BY example

- List all users, sort them by popularity (descending) and name (ascending)

```
SELECT uid, name, age, pop
FROM User
ORDER BY pop DESC, name;
```

- ASC is the default option
- Strictly speaking, only output columns can appear in ORDER BY clause (although some DBMS support more)
- Can use sequence numbers instead of names to refer to output columns: ORDER BY 4 DESC, 2;

Outline

- Lecture 2: Intro to the relational model
- Lectures 3 & 4: SQL (1) & (2)
  - Data-definition language (DDL)
  - Data-manipulation language (DML)
    - SELECT-FROM-WHERE statements
    - Set/bag (DISTINCT, UNION/EXCEPT/INTERSECT (ALL))
    - Subqueries (table, scalar, IN, EXISTS, ALL, ANY)
    - Aggregation and grouping (GROUP BY, HAVING)
    - Ordering (ORDER)
    - Outerjoins (and Nulls)
    - INSERT/DELETE/UPDATE
  - Integrity constraints: specify constraints that the data stored in the database must satisfy

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*?

SQL's solution

- A special value NULL
  - For every domain
  - Special rules for dealing with NULL's
- Example: *User* (*<u>uid</u>*, *name*, *age*, *pop*)
  - ⟨789, "Nelson", NULL, NULL⟩

Three-valued logic

TRUE = 1, FALSE = 0, UNKNOWN = 0.5

x AND y = $\min(x, y)$

x OR y = $\max(x, y)$

NOT x = $1 - x$

- Comparing a NULL with another value (including another NULL) using =, >, etc., the result is NULL
- WHERE and HAVING clauses only select rows for output if the condition evaluates to TRUE
    - NULL is not enough
- Aggregate functions ignore NULL, except COUNT(*)

Unfortunate consequences

- Q1a = Q1b?

```
Q1a. SELECT AVG(pop) FROM User;
```

```
Q1b. SELECT SUM(pop)/COUNT(*) FROM User;
```

- Q2a = Q2b?

```
Q2a. SELECT * FROM User;
```

```
Q2b SELECT * FROM User WHERE pop=pop;
```

- Be careful: NULL breaks many equivalences

Another problem

- Example: Who has NULL *pop* values?

```
SELECT * FROM User WHERE pop = NULL;
```

Does not work!

```
(SELEC * 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;
```

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
foo	NULL	789

A full outerjoin between R and S:

- 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

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

- A left outerjoin ($R ⟕ S$) includes rows in $R \bowtie S$ plus dangling R rows padded with NULL's

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

- A right outerjoin ($R ⟖ S$) includes rows in $R \bowtie S$ plus dangling S rows padded with NULL's

Outerjoin syntax

```
SELECT * FROM Group LEFT OUTER JOIN Member
        ON Group.gid = Member.gid;
```

$\approx Group ⟕_{Group.gid=Member.gid} Member$

```
SELECT * FROM Group RIGHT OUTER JOIN Member
        ON Group.gid = Member.gid;
```

$\approx Group ⟖_{Group.gid=Member.gid} Member$

```
SELECT * FROM Group FULL OUTER JOIN Member
        ON Group.gid = Member.gid;
```

$\approx Group ⟗_{Group.gid=Member.gid} Member$

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

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

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

```
SELECT * FROM Group NATURAL JOIN Member;
```

SQL features covered so far

- SELECT-FROM-WHERE statements
- Set and bag operations
- Table expressions, subqueries
- Aggregation and grouping
- Ordering
- NULL’s and outerjoins

☞Next: data modification statements, constraints

INSERT

- Insert one row
  - User 789 joins Dead Putting Society

```
INSERT INTO Member VALUES (789, 'dps');
```

- Insert the result of a query
  - Everybody joins Dead Putting Society!

```
INSERT INTO Member
    (SELECT uid, 'dps' FROM User
    WHERE uid NOT IN (SELECT uid
            FROM Member
            WHERE gid = 'dps'));
```

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

Outline

- Lecture 2: Intro to the relational model
- Lectures 3 & 4: SQL (1) & (2)
  - Data-definition language (DDL)
  - Data-manipulation language (DML)
    - SELECT-FROM-WHERE statements
    - Set/bag (DISTINCT, UNION/EXCEPT/INTERSECT (ALL))
    - Subqueries (table, scalar, IN, EXISTS, ALL, ANY)
    - Aggregation and grouping (GROUP BY, HAVING)
    - Ordering (ORDER)
    - Outerjoins (and Nulls)
    - INSERT/DELETE/UPDATE
  - Integrity constraints: specify constraints that the data stored in the database must satisfy

Types of SQL constraints

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

NOT NULL constraint examples

```
CREATE TABLE User
(uid DECIMAL(3,0) NOT NULL,
 name VARCHAR(30) NOT NULL,
 twitterid VARCHAR(15) NOT NULL,
 age DECIMAL (2,0),
 pop DECIMAL(3,2));
```

```
CREATE TABLE Group
(gid CHAR(10) NOT NULL,
 name VARCHAR(100) NOT NULL);
```

```
CREATE TABLE Member
(uid DECIMAL(3,0) NOT NULL,
 gid CHAR(10) NOT NULL);
```

Key declaration examples

```
CREATE TABLE User
(uid DECIMAL(3,0) NOT NULL PRIMARY KEY,
 name VARCHAR(30) NOT NULL,
 twitterid VARCHAR(15) NOT NULL UNIQUE,
 age DECIMAL (2,0),
 pop DECIMAL(3,2));
```

At most one primary key per table

Any number of UNIQUE keys per table

```
CREATE TABLE Group
(gid CHAR(10) NOT NULL PRIMARY KEY,
 name VARCHAR(100) NOT NULL);
```

```
CREATE TABLE Member
(uid DECIMAL(3,0) NOT NULL,
 gid CHAR(10) NOT NULL,
PRIMARY KEY(uid,gid));
```

This form is required for multi-attribute keys

Incorrect!

```
CREATE TABLE Member
(uid DECIMAL(3,0) NOT NULL PRIMARY KEY,
 gid CHAR(10) NOT NULL PRIMARY KEY,
```

Referential integrity example

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

☞ That is, no "dangling pointers"

User

uid	*name*	…
142	Bart	…
123	Milhouse	…
857	Lisa	…
456	Ralph	…
789	Nelson	…
…	…	…

Member

uid	*gid*
142	dps
123	gov
857	abc
857	gov
456	abc
456	gov
…	…

Group

gid	*name*
abc	…
gov	…
dps	…
…	…

Referential integrity in SQL

- Referenced column(s) must be PRIMARY KEY
- Referencing column(s) form a FOREIGN KEY
- Example

```
CREATE TABLE Member
(uid DECIMAL(3,0) NOT NULL REFERENCES User(uid),
 gid CHAR(10) NOT NULL,
PRIMARY KEY(uid,gid),
FOREIGN KEY (gid) REFERENCES Group(gid));
```

This form is required for multi-attribute foreign keys

```
CREATE TABLE MemberBenefits
(… ..
FOREIGN KEY (uid,gid) REFERENCES Member(uid,gid));
```

Enforcing referential integrity

Example: *Member.uid references User.uid*

- Insert or update a *Member* row so it refers to a non-existent *uid*
  - Reject

User

uid	*name*	*…*
142	Bart	…
123	Milhouse	…
857	Lisa	…
456	Ralph	…
789	Nelson	…
…	…	…

Member

uid	*gid*
142	dps
123	gov
857	abc
857	gov
456	abc
456	gov
000	**gov**

Reject

Enforcing 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)

User

uid	*name*	*…*
142	Bart	…
123	Milhouse	…
857	Lisa	…
~~456~~	~~Ralph~~	~~…~~
789	Nelson	…
…	…	…

Option 1: Reject

Member

uid	*gid*
142	dps
123	gov
857	abc
857	gov
~~456~~	~~abc~~
~~456~~	~~gov~~
…	….

```
CREATE TABLE Member
(uid DECIMAL(3,0) NOT NULL
REFERENCES User(uid)
ON DELETE CASCADE,
…..);
```

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

Enforcing 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)

```
CREATE TABLE Member
(uid DECIMAL(3,0) NOT NULL
REFERENCES User(uid)
ON DELETE SET NULL,
…..);
```

User

uid	*name*	*…*
142	Bart	…
123	Milhouse	…
857	Lisa	…
~~456~~	~~Ralph~~	…
789	Nelson	…
…	…	…

Member

uid	*gid*
142	dps
123	gov
857	abc
857	gov
NULL	abc
NULL	gov
…	….

Option 3: Set NULL

(set all references to NULL)

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)));
```

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)),
 ...);
```

Exercise Question:
How does it differ from a referential integrity constraint (slides 26-27)?

SQL features covered so far

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

☞Next: triggers, views, indexes (lecture 5)