CSE 412 Database Management

Dr. Bharatesh Chakravarthi (BC)

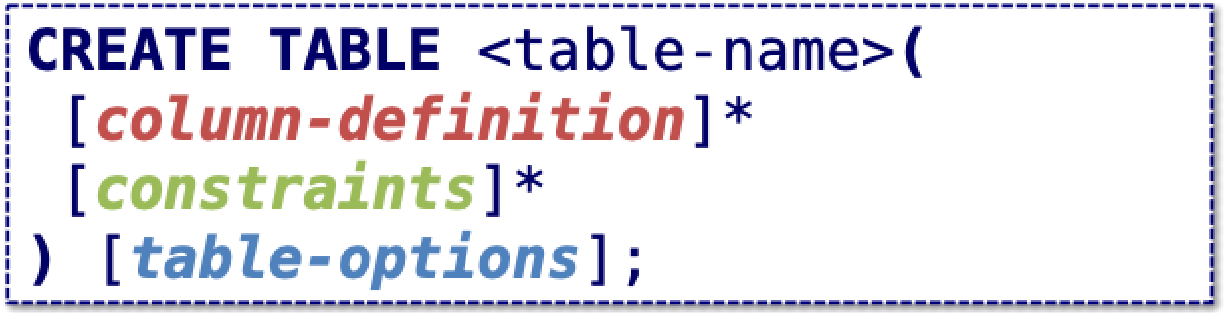
Assistant Teaching Professor

School of Computing and Augmented Intelligence

Arizona State University

Review of SQL Basics

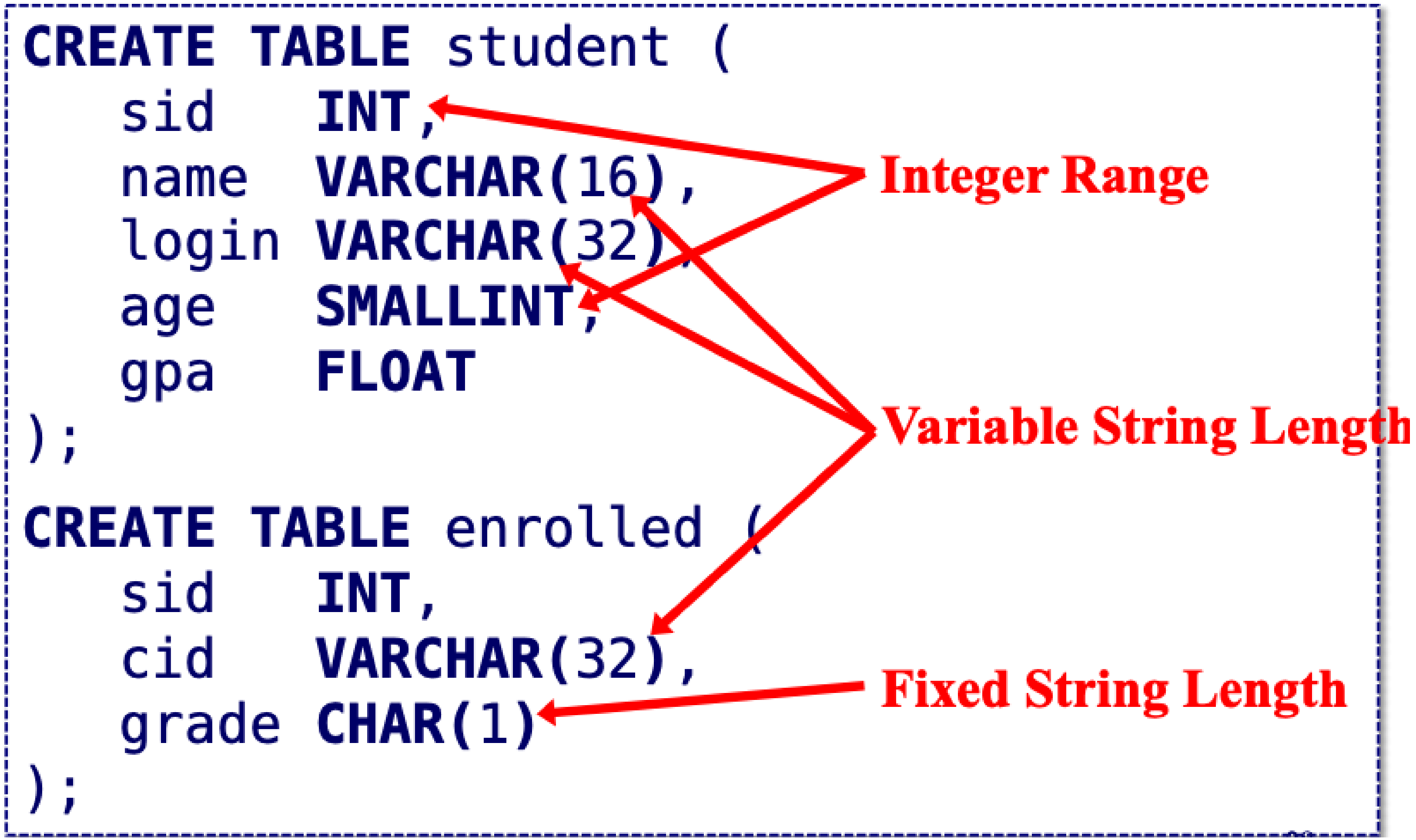
Review of Data Definition Language



```
CREATE TABLE <table-name>(
 [column-definition]*
 [constraints]*
) [table-options];
```

- **Column-Definition:** Comma separated list of column names with types.
- **Constraints:** Primary key, foreign key, and other meta-data attributes of columns.
- **Table-Options:** DBMS-specific options for the table (not SQL-92).

Data Definition Language (DDL)



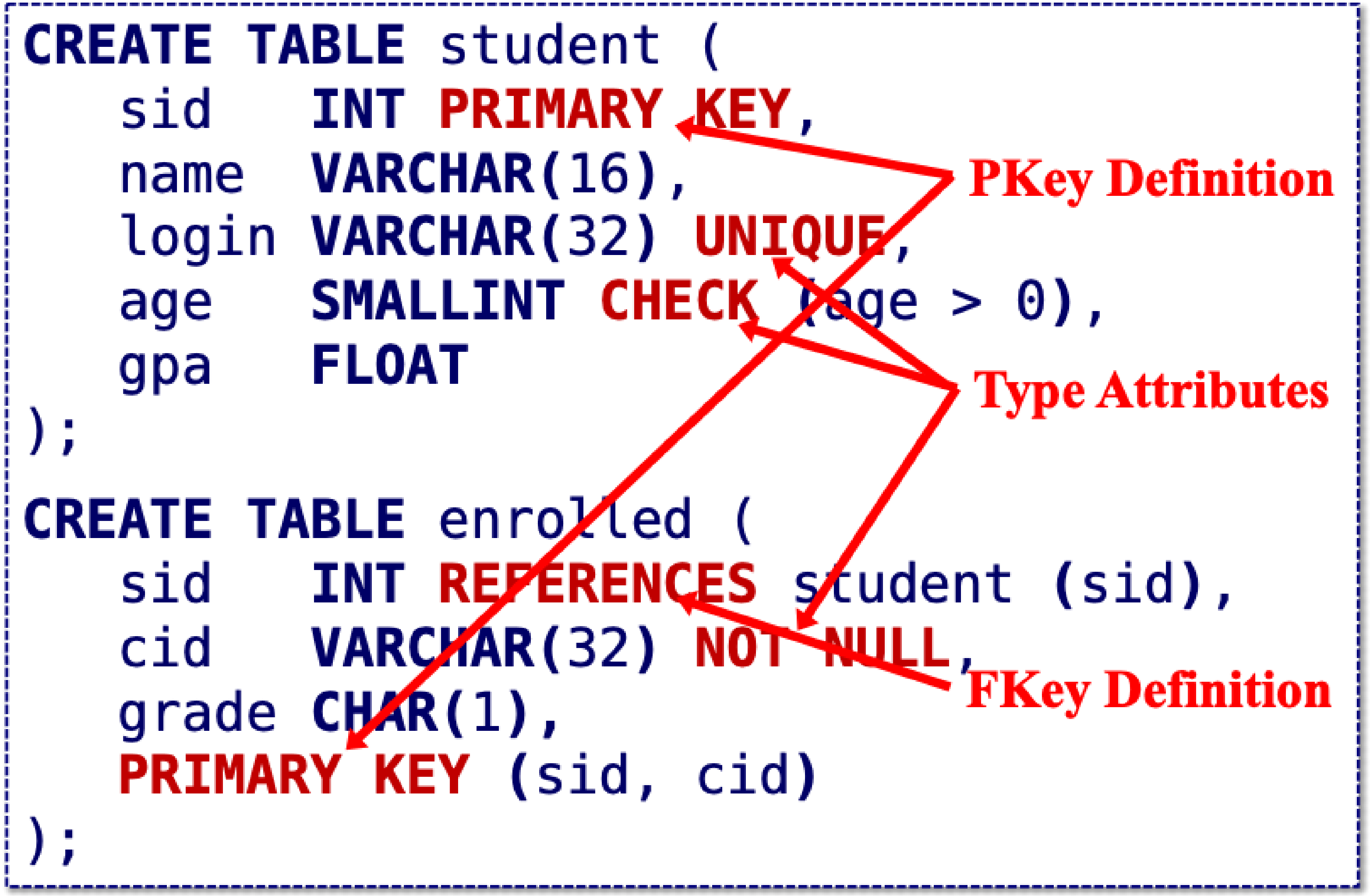
```
CREATE TABLE student (
    sid    INT,
    name   VARCHAR(16),
    login  VARCHAR(32),
    age    SMALLINT,
    gpa    FLOAT
);
CREATE TABLE enrolled (
    sid    INT,
    cid    VARCHAR(32),
    grade  CHAR(1)
);
```

Integer Range

Variable String Length

Fixed String Length

Integrity Constraints



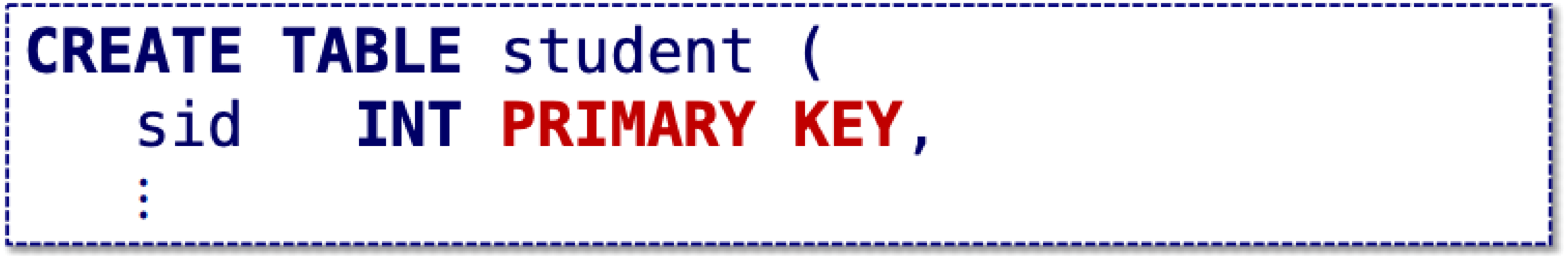
```
CREATE TABLE student (
   sid   INT PRIMARY KEY,
   name  VARCHAR(16),
   login VARCHAR(32) UNIQUE,
   age   SMALLINT CHECK (age > 0),
   gpa   FLOAT
);

CREATE TABLE enrolled (
   sid   INT REFERENCES student (sid),
   cid   VARCHAR(32) NOT NULL,
   grade CHAR(1),
   PRIMARY KEY (sid, cid)
);
```

PKey Definition
Type Attributes
FKey Definition

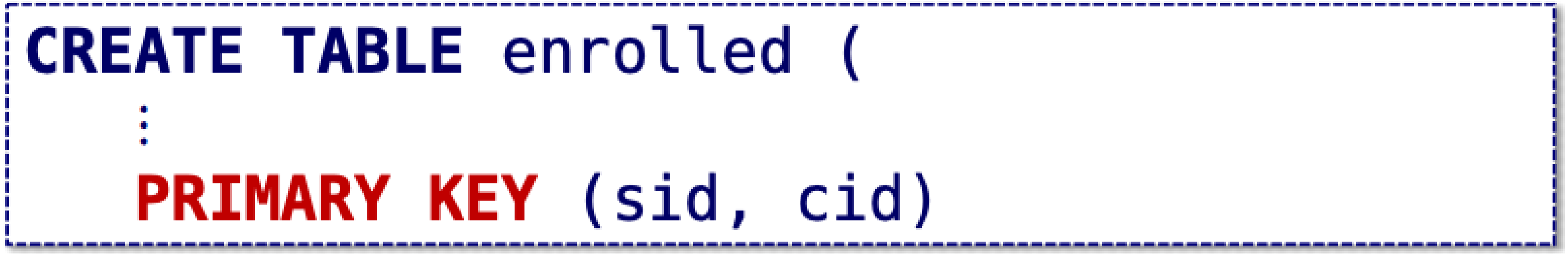
Primary Keys

- Single-column primary key:



```
CREATE TABLE student (
    sid   INT PRIMARY KEY,
    ⋮
```

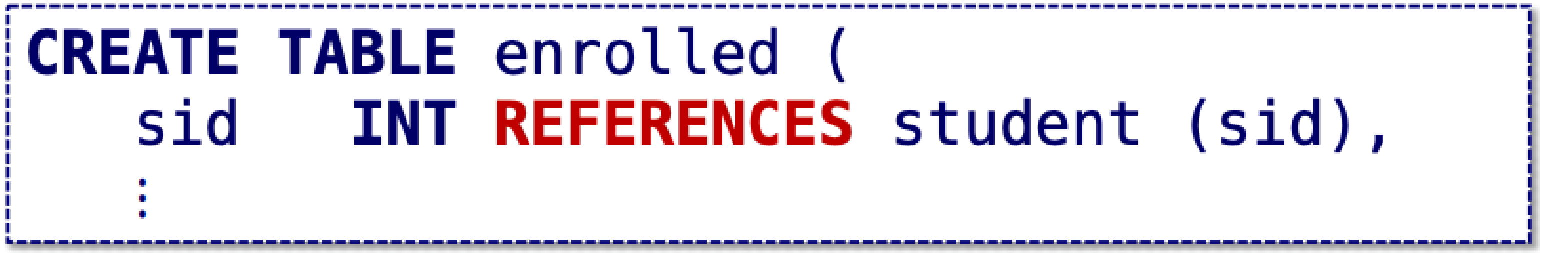
- Multi-column primary key:



```
CREATE TABLE enrolled (
    ⋮
    PRIMARY KEY (sid, cid)
```

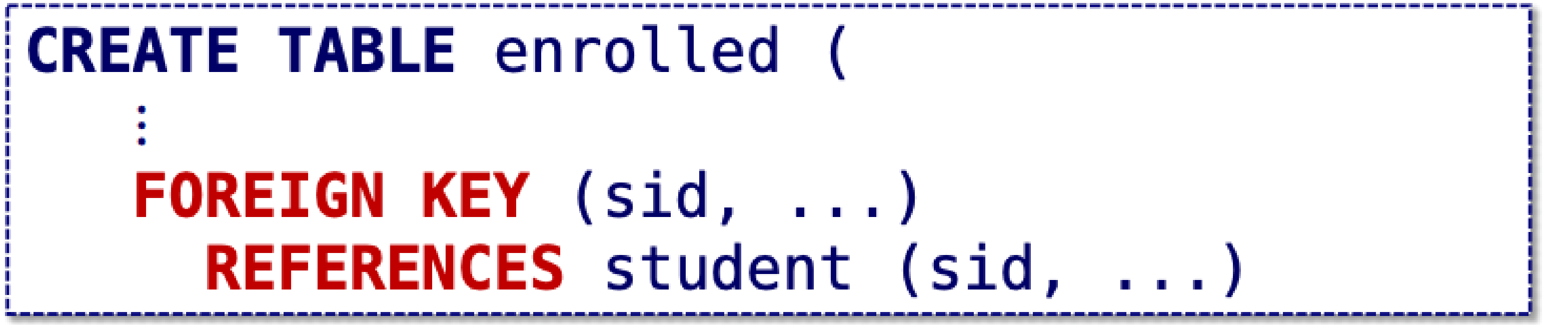
Foreign Key References

- Single-column reference:



```
CREATE TABLE enrolled (
    sid    INT REFERENCES student (sid),
    ⋮
```

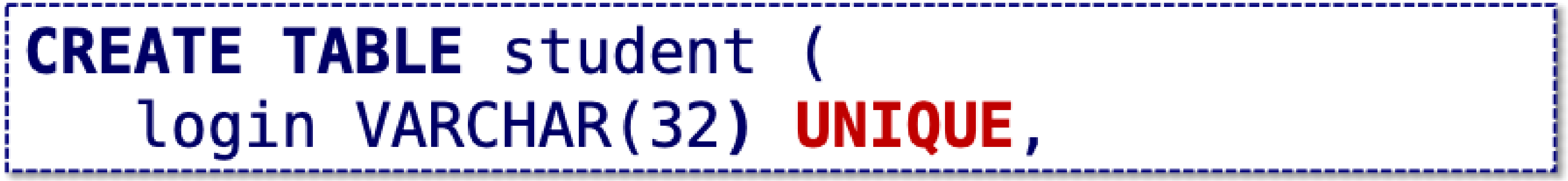
- Multi-column reference:



```
CREATE TABLE enrolled (
    ⋮
    FOREIGN KEY (sid, ...)
      REFERENCES student (sid, ...)
```

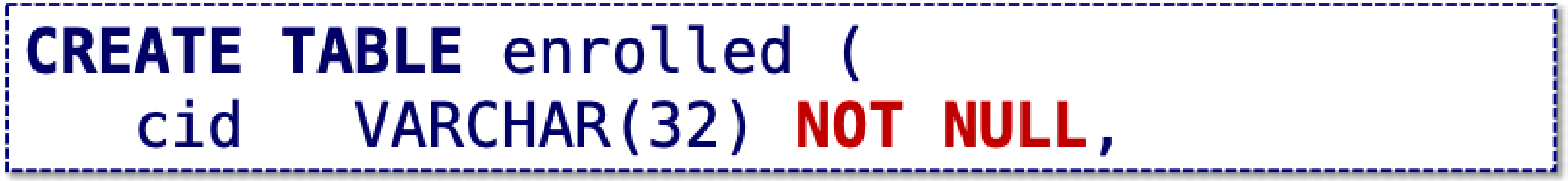
Value Constraints

- Ensure one-and-only-one value exists:



```
CREATE TABLE student (
    login VARCHAR(32) UNIQUE,
```

- Make sure a value is not null:



```
CREATE TABLE enrolled (
    cid   VARCHAR(32) NOT NULL,
```

Tuple- and Attribute- based **Checks**

- Associated with a ***single table***
- Only checked ***when a tuple/attribute is inserted/updated***
  - **Reject** if the 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 the User is modified

Review of Data Manipulation Language

- SELECT A_1, A_2, …, A_n
  FROM R_1, R_2, …, R_m
  WHERE condition;
- Also called as SPJ (selection-projection-join) query
- Can correspond to (but not exactly equivalent to) relational algebra query:

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

Why SFW statements

- Out of many possible ways of structuring SQL statements, *why did the designers choose SELECT-FROM-WHERE*?
  - A large number of queries can be written using only selection, projection, and cross-product (or join)
  - Any query that uses only these operators can be written in a canonical form:

    $$\pi_L\left(\sigma_p(R_1\times\cdots\times R_m)\right)$$

  - SELECT-FROM-WHERE captures this canonical form

Example Database

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
…	…	…	…

Member

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

Group

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

Example: reading a table

- SELECT * FROM User;
  - Single-table query, so no cross product here
  - WHERE clause is optional
  - * is a short hand for “all columns”

Example: selection and projection

- Name of users under 18
  - SELECT name
    FROM User
    WHERE age < 18
- When was Lisa born?
  - SELECT 2025-age
    FROM User
    WHERE name = 'Lisa’;
  - SELECT list can contain expressions
    - Can also use built-in functions such as SUBSTR, ABS, etc.
  - String literals (case sensitive) are enclosed in single quotes

Example: join

- ID’s and names of groups with a user whose name contains “Simpson”

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

Member

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

Group

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

Example: join

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
…	…	…	…

Member

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

- ID’s and names of groups with a user whose name contains “Simpson”
  - SELECT Group.gid, Group.name
    FROM User, Member, Group
    WHERE User.uid = Member.uid
    AND Member.gid = Group.gid
    AND User.name LIKE '%Simpson%’;
  - **LIKE** matches a string against a pattern
  - % matches any sequence of zero or more characters
  - Okay to omit table_name in table_name.column_name if column_name is unique

Group

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

SQL Intermediate

Today’s Agenda

- Data Definition Language
- **Data Manipulation Language**
  - Basic Queries (SELECT-FROM-WHERE)
  - **ORDER BY**

Order By

- SELECT … FROM … WHERE … ORDER BY output_column [ASC|DESC], …;

(In the future: SELECT [DISTINCT] … FROM … WHERE … GROUP BY … HAVING … ORDER BY output_column [ASC|DESC], …;)

- ASC = ascending, DESC = descending
- **Semantics:** After the **SELECT** list has been computed and optional duplicate elimination has been carried out***, sort the output according to ORDER BY specification***

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
…	…	…	…

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 the 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;

Today’s Agenda

- Data Definition Language
- **Data Manipulation Language**
  - Basic Queries (SELECT-FROM-WHERE)
  - ORDER BY
  - **Set Operations**

Forcing set semantics

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
…	…	…	…

Member

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

Group

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

- IDs 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;

    (finds pairs of members who belong to the same group.)

  - *Say Lisa and Ralph are in both the book club and the student government*

*Forcing set semantics means ensuring that a query returns **unique results** by explicitly removing duplicates, typically using operations like DISTINCT, rather than allowing duplicate entries in the result set*

- SELECT DISTINCT m1.uid AS uid1, m2.uid AS uid2 …

  - With DISTINCT, all duplicate (uid1, uid2) pairs are removed from the output

Set Semantics of SFW

- SELECT [DISTINCT] E_1, E_2, …, E_n
  FROM R_1, R_2, …, R_m
  WHERE condition;

SQL set

- 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

Examples of set operations

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

Examples of set operations

- Poke (uid1, uid2, timestamp)
  - (SELECT uid1 FROM Poke)
    EXCEPT
    (SELECT uid2 FROM Poke);
    - **Users who poked others but never got poked by others**

- The EXCEPT operator returns the **set difference** between the two subqueries.
- It gives you all the uid1 values (users who poked others) that are **not** present in the uid2 values (users whom someone poked).

Today’s Agenda

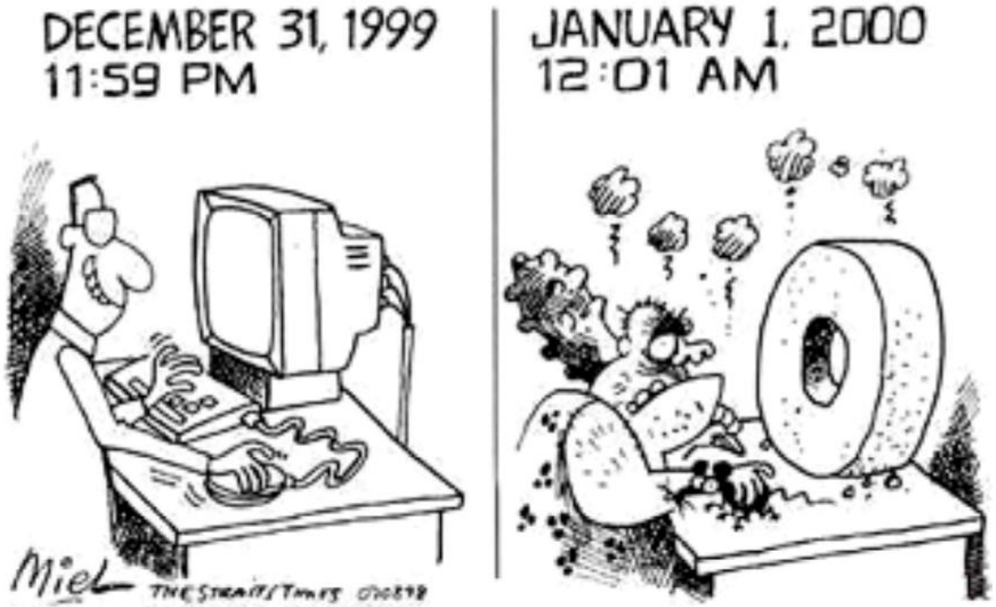
- Data Definition Language
- **Data Manipulation Language**
  - Basic Queries (SELECT-FROM-WHERE)
  - ORDER BY
  - Set Operations
  - **Null Values**

Missing Information

- Example: User (<u>uid</u>, 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?

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 (using a flag)

- 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)
  - Still complicates schema and queries
    - How to get all information about users in a table?
    - Natural join doesn’t work!

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 NULLs

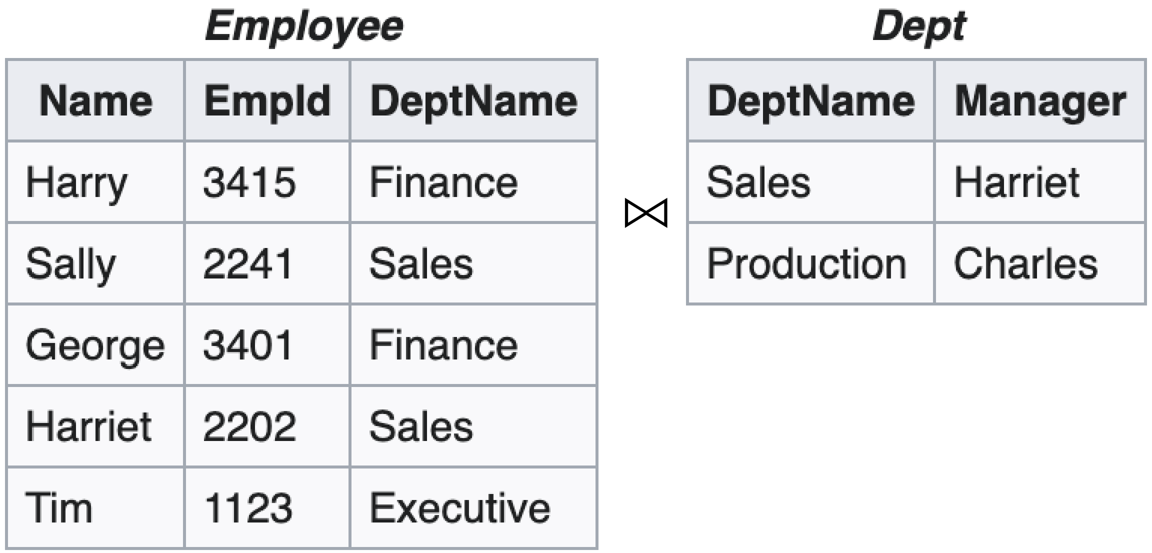
- When we operate on a NULL and another value (including another NULL) using +, −, etc., the result is NULL
- Aggregate functions ignore NULL, except COUNT(*) (since it counts rows)

IS NULL/IS NOT NULL

- 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;

Outerjoin motivation

First let’s see: what is the natural join result?



Employee

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales
Tim	1123	Executive

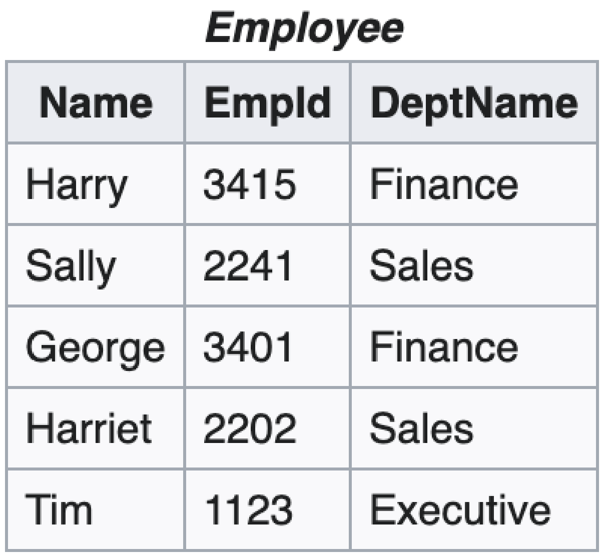
$\bowtie$

Dept

DeptName	Manager
Sales	Harriet
Production	Charles

Outerjoin motivation

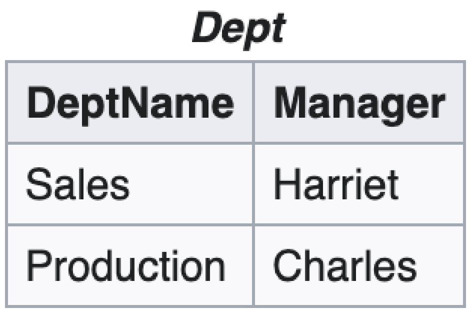
First let’s see: what is the natural join result?



Employee

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales
Tim	1123	Executive

⋈



Dept

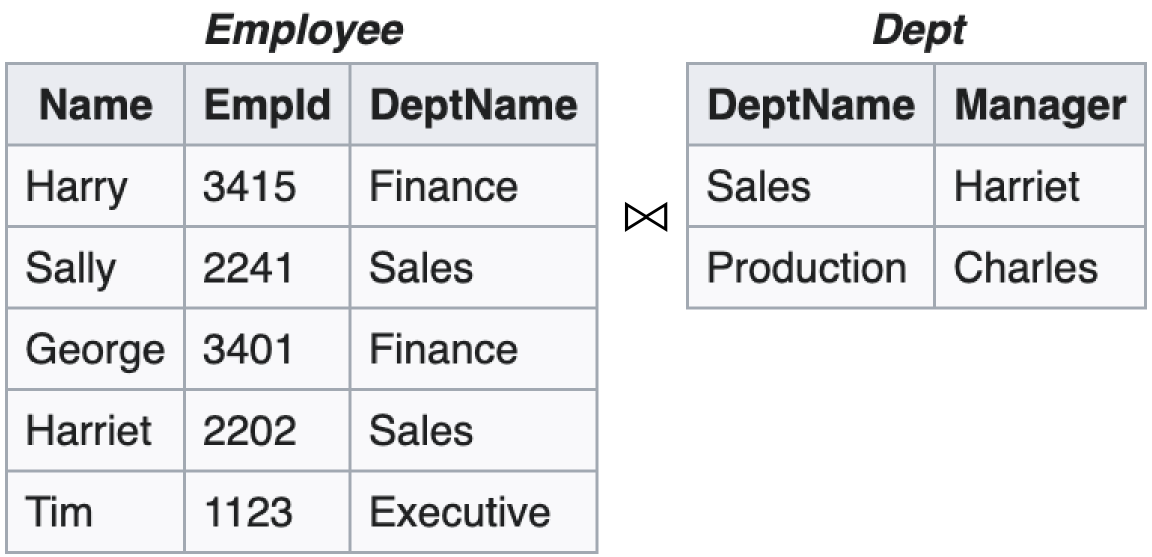
DeptName	Manager
Sales	Harriet
Production	Charles

=

Name	EmpId	DeptName	Manager

Outerjoin motivation

First let’s see: what is the natural join result?



Employee

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales
Tim	1123	Executive

⋈

Dept

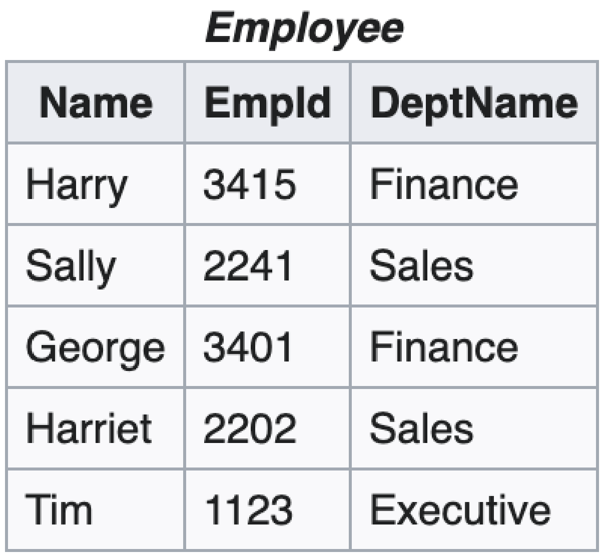
DeptName	Manager
Sales	Harriet
Production	Charles

=

Name	EmpId	DeptName	Manager
Sally	2241	Sales	Harriet
Harriet	2202	Sales	Harriet

Outerjoin motivation

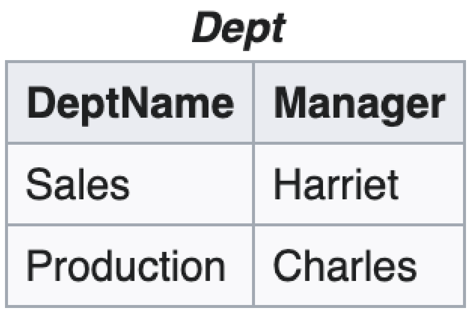
First let’s see: what is the natural join result?



Employee

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales
Tim	1123	Executive

$\bowtie$



Dept

DeptName	Manager
Sales	Harriet
Production	Charles

=

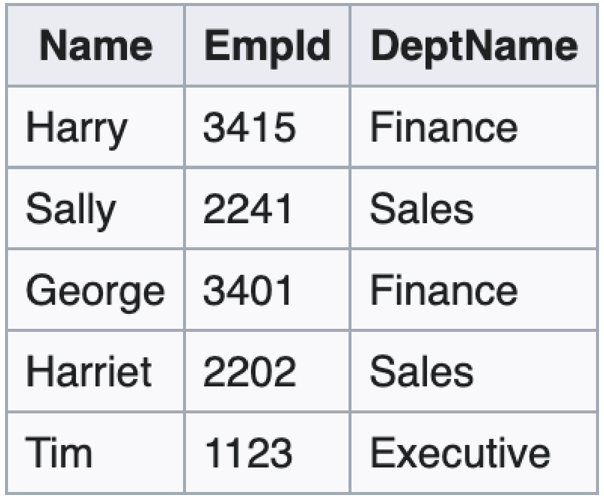
Name	EmpId	DeptName	Manager
Sally	2241	Sales	Harriet
Harriet	2202	Sales	Harriet

Bad, other employee information get lost, because their departments do not have a record in the Dept table

Optional: Left Outer Natural Join

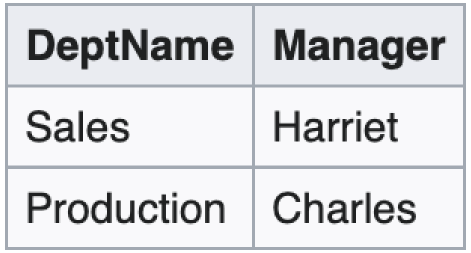
- In a left outer join, Employee rows without a matching Department row appear in the result, but not vice versa.

Employee



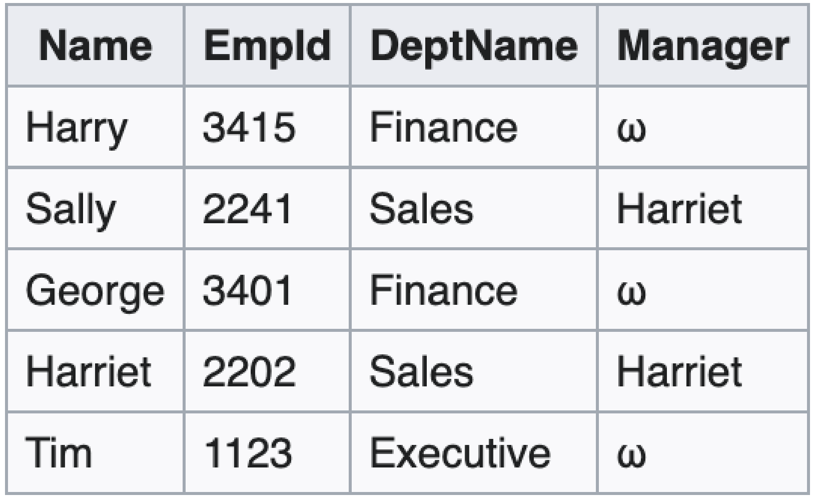
Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales
Tim	1123	Executive

Dept



DeptName	Manager
Sales	Harriet
Production	Charles

Employee ⟕ Dept



Name	EmpId	DeptName	Manager
Harry	3415	Finance	ω
Sally	2241	Sales	Harriet
George	3401	Finance	ω
Harriet	2202	Sales	Harriet
Tim	1123	Executive	ω

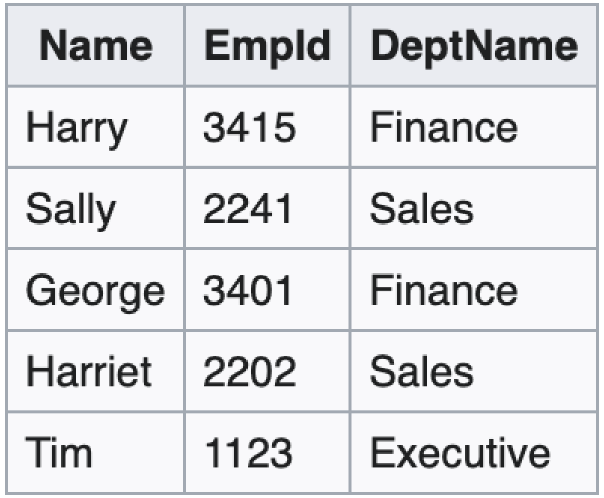
Optional: Left Outer Natural Join

- ⟕

$$(R \bowtie S) \cup ((R - \pi_{r_1, r_2, \ldots, r_n}(R \bowtie S)) \times \{(\omega, \ldots \omega)\})$$

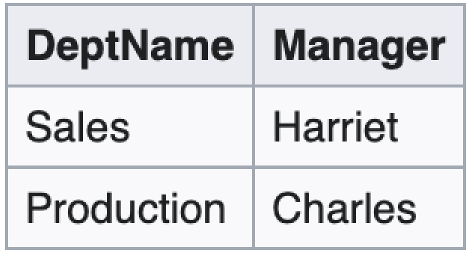
- In a left outer join, Employee rows without a matching Department row appear in the result, but not vice versa.

Employee



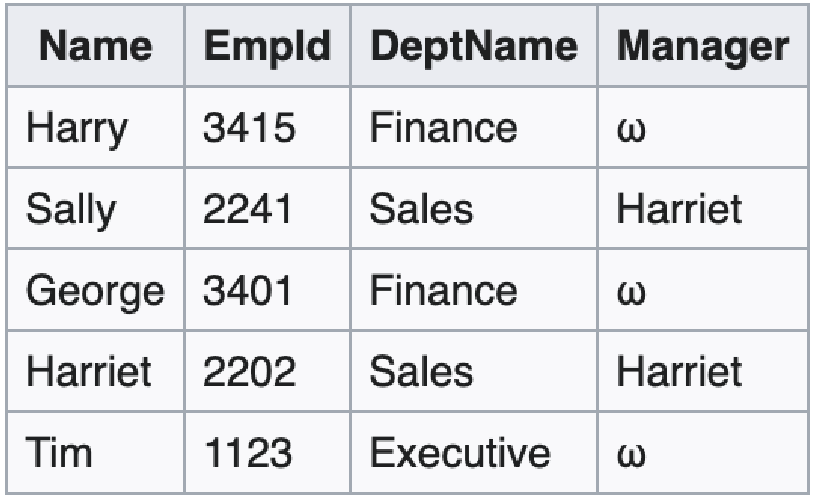
Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales
Tim	1123	Executive

Dept



DeptName	Manager
Sales	Harriet
Production	Charles

Employee ⟕ Dept



Name	EmpId	DeptName	Manager
Harry	3415	Finance	ω
Sally	2241	Sales	Harriet
George	3401	Finance	ω
Harriet	2202	Sales	Harriet
Tim	1123	Executive	ω

SQL Continued..

- Data Definition Language
- **Data Manipulation Language**
  - Basic Queries (SELECT-FROM-WHERE)
  - ORDER BY
  - Set Operations
  - Null Values
  - **Aggregation**

Aggregates

- Standard SQL aggregate functions: COUNT, SUM, AVG, MIN, MAX
- Example: number of users under 18, and their average popularity
  - SELECT COUNT(*)
    FROM User
    WHERE age < 18;
  - COUNT(*) counts the number of rows

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

- 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
  ```

Example of Grouping 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

Example of Aggregates (with no Group By)

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

SELECT AVG(pop) AS 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

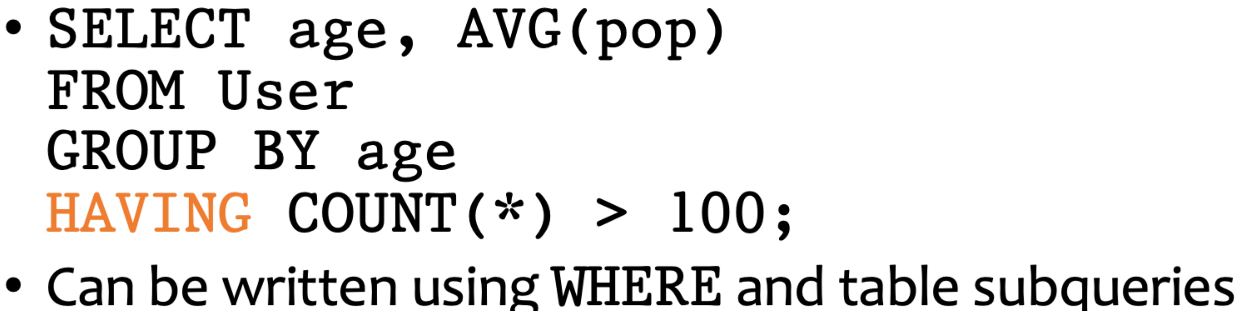
Having

- Used to filter groups based on the group properties (e.g., aggregate values, GROUP BY column values)
- SELECT … FROM … WHERE … GROUP BY …

  HAVING condition;

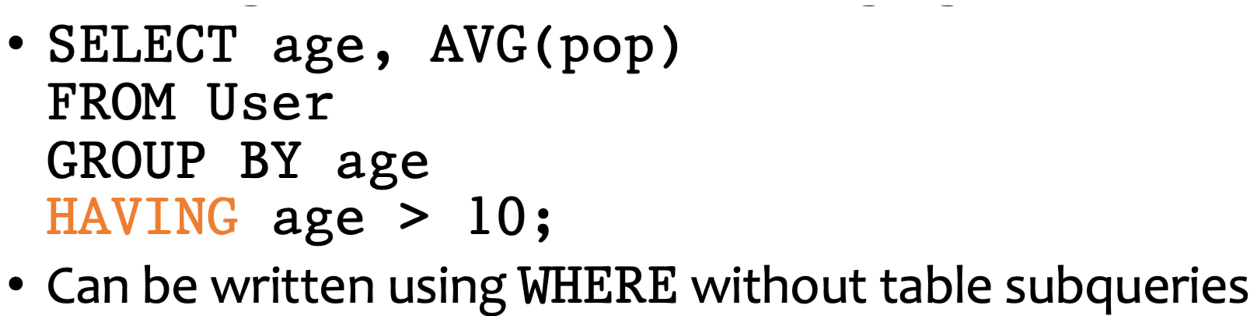
Having examples

- 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

- 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