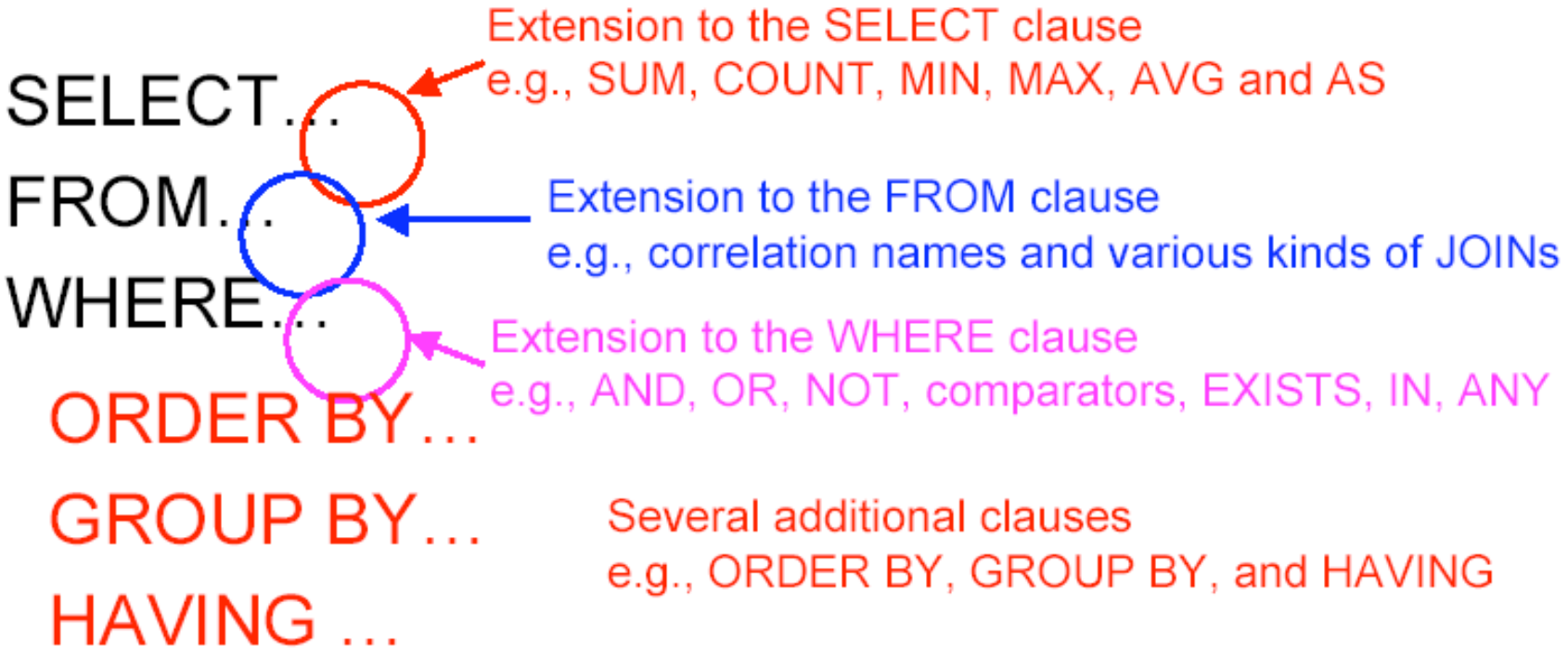
Review of Last Lecture

- Extension to Where
  - Boolean and comparator
  - Pattern and LIKE, NOT LIKE
    - %, _
  - Subquery
    - How compare an attribute with a table of values generated by subquery?
      - IN and NOT IN
      - EXISTS, UNIQUE, NOT EXISTS, NOT UNIQUE
      - Comparator ANY, ALL

Today's Plan

- Order By
  - ASC
  - DESC
- Group By, Having
  - Special requirements
- Union, intersect, minus
  - Special requirements
- Effect of NULL

Today's Plan


SELECT…
FROM…
WHERE…
ORDER BY…
GROUP BY…
HAVING …
Extension to the SELECT clause
e.g., SUM, COUNT, MIN, MAX, AVG and AS
Extension to the FROM clause
e.g., correlation names and various kinds of JOINs
Extension to the WHERE clause
e.g., AND, OR, NOT, comparators, EXISTS, IN, ANY
Several additional clauses
e.g., ORDER BY, GROUP BY, and HAVING

ORDER BY

Sort the result on one or more attributes
Can specify ASC, DESC

```
SELECT      FName, LName
FROM        Employee
ORDER BY    LName       /* The default is ASC */
                        /* Don't be lazy! */
                        /* Specify it to avoid confusion */
```

```
SELECT      FName, LName
FROM        Employee
ORDER BY    Salary DESC
```

GROUP BY

```
SELECT      DNO, COUNT(*)      /* Count # of E in each D */
FROM        Employee
GROUP BY    DNO
```

```
SELECT      Sex, COUNT(*)      /* Find # of E in each gender */
FROM        Employee
GROUP BY    Sex
```

Any form of SQL query (e.g., with or without subqueries) can have the answer "grouped". When a query answer is grouped, there is one output row - for each group.

Example of Computing GROUP BY

```
SELECT age, AVG(GPA) FROM Student GROUP BY age;
```

SID	*name*	*age*	*GPA*
142	Bart	10	2.3
857	Lisa	8	4.3
123	Milhouse	10	3.1
456	Ralph	8	2.3
...	...	...	...

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

SID	*name*	*age*	*GPA*
142	Bart	10	2.3
123	Milhouse	10	3.1
857	Lisa	8	4.3
456	Ralph	8	2.3
...	...	...	...

Compute SELECT for each group

age	*AVG_GPA*
10	2.7
8	3.3
...	...

Restriction on SELECT With Aggregation/Group

- If a query uses aggregation/group by, then every attribute referenced in SELECT MUST be either
  - Grouping attribute or
  - Aggregated

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

Illegal Query Example

- You might think you could find the employees with the lowest salary:

```
SELECT    SSN, MIN(Salary)
FROM      Employee
```

- But this query is illegal in SQL.
  - SQLite allows it and selects the first row matches (the first employee in the table with the lowest salary). But what if there are more than employees with the same lowest salary? Only the first one will be shown!

More Examples of Invalid Queries

- `SELECT ~~SID,~~ age FROM Student GROUP BY age;`
  - Recall there is one output row per group
  - There can be multiple SID values per group
- `SELECT ~~SID,~~ MAX(GPA) FROM Student;`
  - Recall there is only one group for an aggregate query with no `GROUP BY` clause
  - There can be multiple SID values
  - Wishful thinking (that the output SID value is the one associated with the highest GPA) does NOT work

Why Wrong

```
SELECT SID, age FROM Student GROUP BY age
```

SID	name	age	GPA
142	Bart	10	2.3
857	Lisa	8	4.3
123	Milhouse	10	3.1
456	Ralph	8	2.3
...	...	...	...

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

SID	name	age	GPA
142	Bart	10	2.3
123	Milhouse	10	3.1
857	Lisa	8	4.3
456	Ralph	8	2.3
...	...	...	...

Why Right

```
SELECT age, AVG(GPA) FROM Student GROUP BY age;
```

SID	*name*	*age*	*GPA*
142	Bart	10	2.3
857	Lisa	8	4.3
123	Milhouse	10	3.1
456	Ralph	8	2.3
...	...	...	...

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

SID	*name*	*age*	*GPA*
142	Bart	10	2.3
123	Milhouse	10	3.1
857	Lisa	8	4.3
456	Ralph	8	2.3
...	...	...	...

HAVING

```
SELECT      Sex                  /* Find gender group above */
FROM        Employee             /* minimal salary $25K  */
GROUP BY    Sex
HAVING      avg(Salary) > 25000
```

The HAVING clause is a predicate evaluated against each group. A group participates in the query answer if it satisfies the HAVING predicate

HAVING: Problem

```
SELECT      DNO
FROM        Employee
GROUP BY    DNO
HAVING      COUNT(*) > 3
```

	dno
1	1
2	4
3	5
4	6
5	7
6	8

Employee

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring,TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

Requirements on HAVING Conditions

- HAVING may refer to attributes of relations in the FROM clause, as long as the attribute makes sense within a group; i.e., it is either:
  - A grouping attribute, or
  - Aggregated

- Recall: if GROUP BY is used or aggregation is used, all attributes in SELECT must be either grouping attributes or aggregated attributes.

HAVING: Exercise

- From Sells(bar, beer, price) and Beers(name, manf), find the average price of those beers that are either served in at least three bars or are manufactured by Pete's.

HAVING: Solution

Beer groups with at least 3 non-NULL bars and also beer groups where the manufacturer is Pete's.

```
SELECT beer, AVG(price)
FROM Sells
GROUP BY beer
HAVING COUNT(bar) >= 3 OR
    beer IN (SELECT name
             FROM Beers
             WHERE manf = 'Pete''s');
```

Beers manufactured by Pete's.

GROUP BY and Having: Exercise

- For each department, retrieve the department number, the number of employees who work in the department and their average salary.

```
SELECT    DNO, COUNT(*), AVG(SALARY)
FROM      EMPLOYEE
```

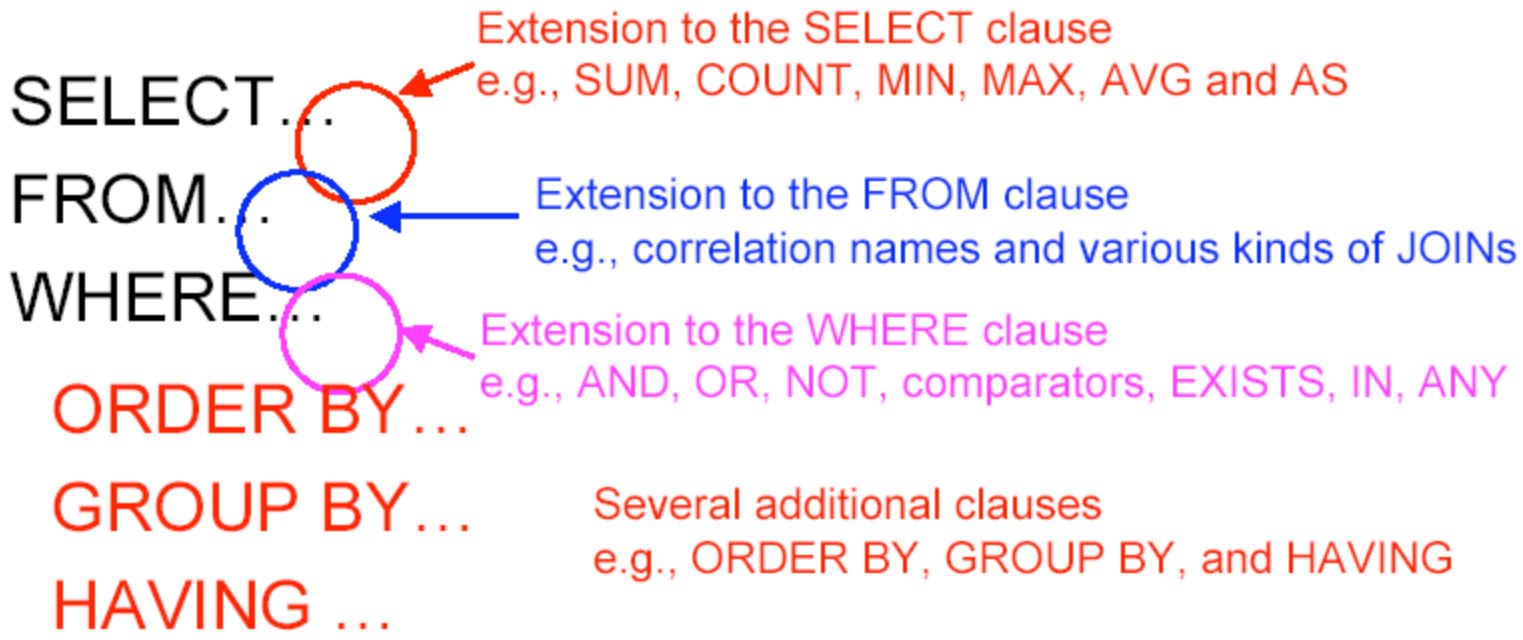
- For each project on which more than one employee works, retrieve the project number, the project name, and the number of employees work on the project

```
SELECT    PNUMBER, PNAME, COUNT(*)
FROM      PROJECT, WORK_ON
WHERE     PNUMBER = PNO
```

Evaluation Order

- 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

Combining Queries



(SELECT…FROM…WHERE…)
UNION
(SELECT…FROM…WHERE…)

And operators that expect two or more complete SQL queries as operands
e.g., UNION and INTERSECT

UNION

```
Query1
        UNION [ALL]
Query2
```

```
SELECT      SUPERSSN
FROM        Employee
        UNION
SELECT      MGRSSN
FROM        Department;
```

UNION eliminates duplicate rows, while
UNION ALL does not

Exercise: how many rows in the output?

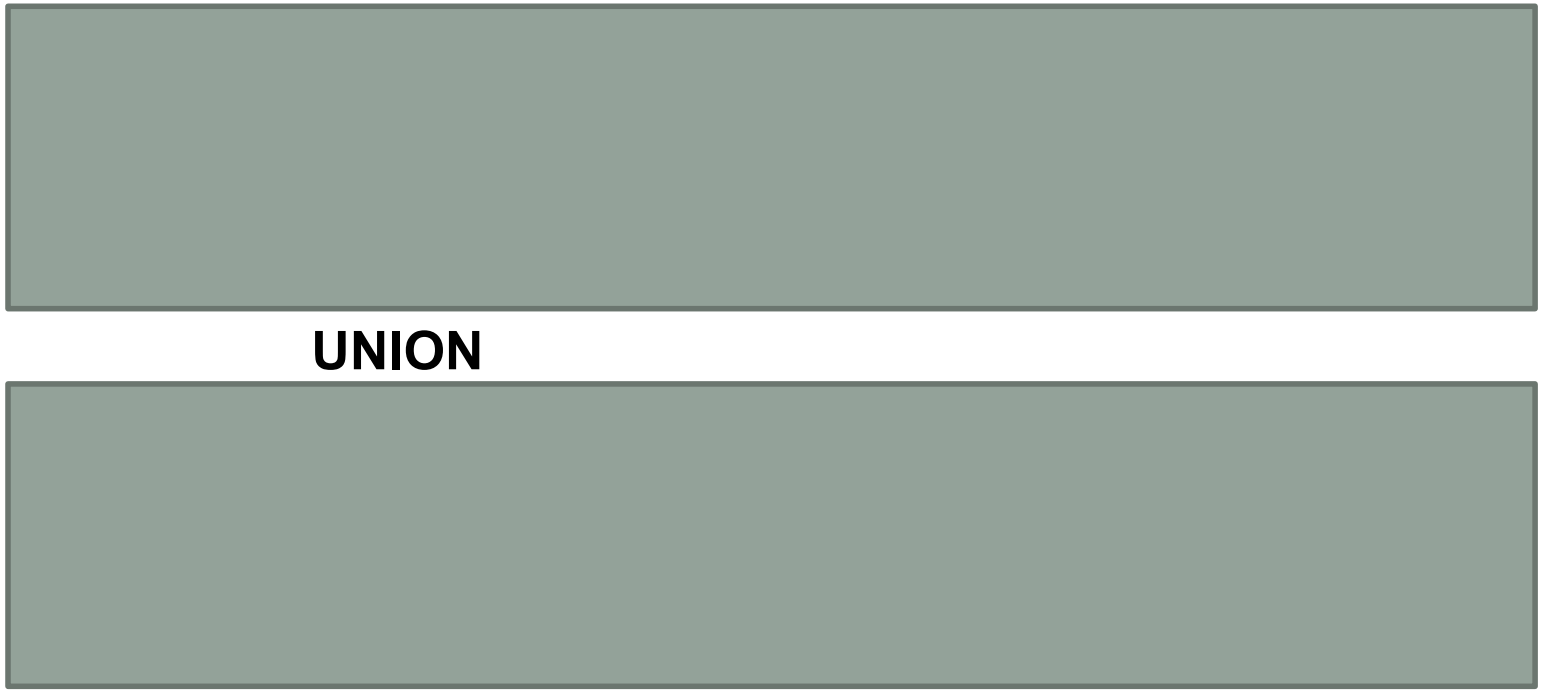
```
SELECT      FName
FROM        Employee;
```

```
SELECT DISTINCT FName
FROM            Employee;
```

```
SELECT      FName
FROM        Employee
      UNION
SELECT      FName
FROM        Employee;
```

UNION: Exercise

- Make a list of all project numbers for projects that involve an employee whose last name is 'Smith' as a worker or as a manager of the department that controls the project.



INTERSECT

```
Query1
        INTERSECT
Query2
```

```
SELECT      SUPERSSN
FROM        Employee
        INTERSECT
SELECT      MGRSSN
FROM        Department;
```

INTERSECT eliminates duplicate rows.

MINUS (DIFFERENCE, EXCEPT)

```
Query1
        EXCEPT
Query2
```

```
SELECT      SUPERSSN
FROM        Employee
        EXCEPT
SELECT      MGRSSN
FROM        Department;
```

In SQLite, the keyword is **EXCEPT**

Union Compatible

- Just like in RA, queries combined by UNION, INTERSECT and MINUS must be union compatible (type compatible).
  - Same number of attributes and same domain for corresponding attributes.

In-Class Exercise #1

- For each department, retrieve the department number, name of the department manager, the number of employees who work in the department and their average salary.
- Hint: imagine that there is an E table and a M table.

What is the output?

```
/* The employee table has 40 employees. */
SELECT      Count(*)
FROM        Employee
```

```
SELECT      Count(*)
FROM        Employee
WHERE       NULL = NULL
```

```
SELECT      Count(*)
FROM        Employee
WHERE       NULL != NULL
```

NULL Value

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The predicate **is null** can be used to check for null values.
  - E.g. Find all loan number which appear in the *loan* relation with null values for *amount*.

    select *loan-number*
    from *loan*
    where *amount* **is null**
- The result of any arithmetic expression involving *null* is *null*
  - E.g. 5 + null returns null
- However, aggregate functions simply ignore nulls
  - more on this shortly

NULL: Example

- Retrieve the names of all employees who do not have supervisors.

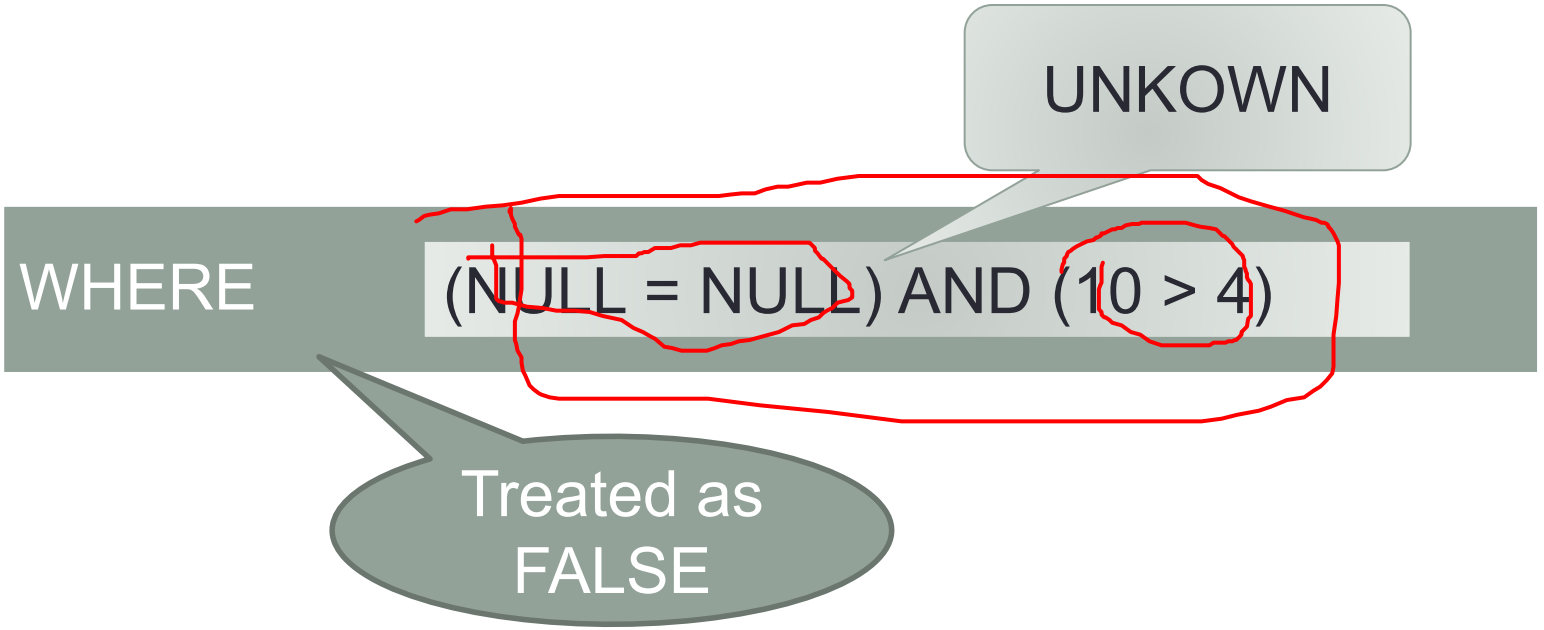
```
SELECT   FNAME, LNAME
FROM     EMPLOYEE
WHERE    SUPERSSN  IS  NULL
```

Three Valued Logic

- Any comparison with *null* returns *unknown*
  - *E.g. 5 < null or null <> null or null = null*
- Three-valued logic using the truth value *unknown*:
  - OR: (*unknown* **or** *true*) = *true*, (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown*) = *unknown*
  - AND: *(true* **and** *unknown)* = *unknown, (false* **and** *unknown)* = *false,*
    (unknown **and** *unknown)* = *unknown*
  - NOT: *(***not** *unknown)* = *unknown*
  - "*P* **is unknown"** evaluates to true if predicate *P* evaluates to *unknown*
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*

WHERE and UNKNOWN

- If a Boolean expression is evaluated to UNKNOWN, WHERE treats it as FALSE.
- But UNKNOWN is not the same as FALSE.



NULL and Aggregation

- NULL never contributes to a sum, average, or count, and can never be the minimum or maximum of a column.
  - NULL is simply ignored.
- But if there are no non-NULL values in a column, then the result of the aggregation is NULL.

- The only exception is count(*): count the number of rows regardless of NULL or not.

Unfortunate Consequence

- `SELECT AVG(GPA) FROM Student;`
  `SELECT SUM(GPA)/COUNT(*) FROM Student;`
  - Not equivalent
  - Although `AVG(GPA) = SUM(GPA)/COUNT(GPA)` still
- `SELECT * FROM Student;`
  `SELECT * FROM Student WHERE GPA = GPA;`
  - Not equivalent

☞ Be careful: `NULL` breaks many equivalences

Surprising Example

- From the following Sells relation:

bar	beer	price

```
SELECT bar
FROM Sells
WHERE price < 2.00 OR price >= 2.00;
```

```
SELECT bar
FROM Sells
```

UNKNOWN UNKNOWN

UNKNOWN

Example: Effect of NULL's

```
SELECT count(*)
FROM Sells
WHERE beer = 'Bud';
```

The number of bars that sell Bud.

```
SELECT count(price)
FROM Sells
WHERE beer = 'Bud';
```

The number of bars that sell Bud at a known price.

SQL Example

- Find name of employee who is a supervisor and a department manager.

```
SELECT    DISTINCT E.FName, E.LName
FROM      Employee E, Employee S, Department D
WHERE     (E.SSN = S.Superssn)
          AND (E.SSN = D.Mgrssn);
```

In-class Exercise #2

- Project the total cost if the company gives a 5% raise to all supervisors. (be careful with duplicates!)

In-class Exercise #3

- Project the total cost if the company gives a 5% raise to all managers and all supervisors. (be careful with duplicates!)

Summary of SQL Queries I

- A query in SQL can consist of up to six clauses, but only the first two, SELECT and FROM, are mandatory. The clauses are specified in the following order:

SELECT <attribute list>
FROM <table list>
[WHERE <condition>]
[GROUP BY <grouping attribute(s)>]
[HAVING <group condition>]
[ORDER BY <attribute list>]

Summary of SQL Queries II

- The SELECT-clause lists the attributes or functions to be retrieved
- The FROM-clause specifies all relations (or aliases) needed in the query but not those needed in nested queries
- The WHERE-clause specifies the conditions for selection and join of tuples from the relations specified in the FROM-clause
- GROUP BY specifies grouping attributes
- HAVING specifies a condition for selection of groups
- ORDER BY specifies an order for displaying the result of a query
- A query is evaluated by first looking up the tables in the FROM-clause, then applying the WHERE-clause, then GROUP BY and HAVING, and finally the SELECT-clause

Summary of Today's Lecture

- Order By
  - ASC
  - DESC
- Group By, Having
  - Special requirements
- Union, intersect, minus
  - Special requirements
- Effect of NULL