CSE 180, Database Systems I
Shel Finkelstein, UC Santa Cruz
Lecture 4

Chapter 3: Introduction to SQL, Part 2

Database System Concepts, 7th Edition.

Modified by Shel Finkelstein for CSE 180

Important Notices

- Lab2 assignment has been posted on Piazza under Resources→Lab2. See the Piazza announcement about Lab2, **as well as Query 1 correction**.
  - For Lab2, you must write a revised create.sql file that specifies some additional constraints, and you must also write 5 SQL queries.
    - You may lose credit if you use DISTINCT, but it wasn't needed.
    - You will lose credit if you **don't** use DISTINCT, but it was needed.
  - Data loading file has already been posted under Resources→Lab2. Use of that file is not required for Lab2, but it may help you test your solution
    - We won't tell you answers to queries on that data.
    - Your queries need to be correct for all database instances!
  - Lab2 will be discussed at Lab Sections; please go to your Lab Section!
  - Lab2 is due on Canvas as a zip file by **Sunday, February 2, 11:59pm**.
    - Late Lab Assignments will not be accepted.
    - Be sure that you post the correct file!
    - Canvas will be used for both Lab submission and grading.
- The second Gradiance Assignment, "CSE 180 Winter 2020 #2", has been assigned, and is due by **Tuesday, Feb 4, 11:59pm**.
- See Piazza notices about LSS Tutoring with Jeshwanth Bheemanpally.

Important Notices

- Access to all webcasts went away on Wednesday, January 29.
  - Attend Lectures, Lab Sections, Office Hours and LSS Tutoring.
- **Reminder:** Midterm is on **Wednesday, February 12.**
  - No make-ups, no early Midterms, no late Midterms … and no devices.
  - You may bring a **single two-sided 8.5" x 11" sheet of paper** with as much info written (or printed) on it as you can fit and read unassisted.
    - No sharing of these sheets will be permitted.
  - "Practice Midterm" from Fall 2019 has been posted on Piazza under Resources→Exams.
    - We'll be covering more material before Midterm this quarter.
    - Solution will be posted next week … but take it yourself first, rather than just reading the solution.
  - Hope that all requests for DRC accommodation have been submitted.
    - DRC students will receive access to webcasts only if that specific accommodation has been requested by DRC.
  - Piazza announcement will describe required seating pattern for Midterm.

Outline

- Overview of The SQL Query Language
- SQL Data Definition
- Basic Query Structure of SQL Queries
- Additional Basic Operations
- Set Operations
- Null Values
- Aggregate Functions
- Nested Subqueries
- Modification of the Database

Example Relations

- *department(dept_name, building, budget)*
- *course(course_id, title, dept_name, credits)*
- *instructor(ID, name, dept_name, salary)*
- *student(ID, name, dept_name, tot_cred)*
- *section(course_id, sec_id, semester, year, building, room_number, time_slot_id)*
- *teaches(ID, course_id, sec_id, semester, year)*
- *takes(ID, course_id, sec_id, semester, year, grade)*

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value.

  AVG: average value

  MIN: minimum value

  MAX: maximum value

  SUM: sum of values

  COUNT: number of values

Aggregate Functions: Examples

- Find the average salary of instructors in the Computer Science department.
  - **SELECT AVG**(*salary*)
    FROM *instructor*
    WHERE *dept_name* = 'Comp. Sci.';
- Find the total number of instructors who teach a course in the Spring 2010 semester.
  - **SELECT COUNT**(**DISTINCT** *ID*)
    FROM *teaches*
    WHERE *semester* = 'Spring' **AND** *year* = 2018;
- Find the number of tuples in the *course* relation
  - **SELECT COUNT**(*)
    FROM *course*;
- Find the average salary of instructors in the Computer Science department, with a 3% raise.
  - **SELECT AVG**(1.03 * *salary*)
    FROM *instructor*
    WHERE *dept_name* = 'Comp. Sci.';

Aggregate Functions with GROUP BY

- For each department, find the average salary of the instructors in that department.
  - **SELECT** *dept_name*, **AVG**(*salary*) AS *avg_salary*
    FROM *instructor*
    GROUP BY *dept_name*;

ID	*name*	*dept_name*	*salary*
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

dept_name	*avg_salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Aggregation (Cont.)

- Attributes in **SELECT** clause outside of aggregate functions must appear in the **GROUP BY** list.

```
/* erroneous query */
SELECT dept_name, ID, AVG(salary)
FROM instructor
GROUP BY dept_name;
```

Aggregation and Grouping Semantics

SELECT [**DISTINCT**] $A_1, A_2, \ldots, A_m$, AGGOP(…)
FROM $r_1, r_2, \ldots r_m$
[**WHERE** *P*]
[**GROUP BY** <list of grouping attributes>]
[**ORDER BY** < list of attributes/expressions [**asc** | **desc**] >];

If the **SELECT** clause has aggregates AGGOP, then $A_1, A_2, \ldots, A_m$ must come from the list of grouping attributes.

- Let Result denote an empty multiset of tuples.
- For every tuple t_1 from r_1, t_2 from r_2, …, t_m from r_m
  - If t_1, …, t_n satisfy predicate *P (that is*, if P evaluates to true), then add the resulting tuple that consists of A_1, A_2, …, A_m components (including attributes of the AGGOP operators) from the t_i into Result.
- Group the tuples in Result according to list of grouping attributes.
  - If **GROUP BY** is omitted, the entire table is regarded as ONE group.
- Apply aggregate operator(s) on the tuples in each group to get a tuple that is put in Result.
- If **DISTINCT** appears in the **SELECT** clause, then remove duplicates from the Result.
- If **ORDER BY** <list of attributes/expressions> appears, then order the tuples in the Result according to the **ORDER BY** clause.
- Return Result.

Grouping, Aggregation, and Nulls

- **NULL** values are ignored in (almost all) aggregations.
  - They do not contribute to the **SUM, AVG, COUNT, MIN, MAX** of an attribute.
  - **COUNT**(*) = number of tuples in a relation (even if some columns are NULL)
  - **COUNT**(A) is the number of tuples with non-null values for attribute A
  - **COUNT**(**DISTINCT** A) is the number of different values for tuples with non-null values for attribute A

- **SUM, AVG, MIN, MAX** on an empty result (no tuples) is **NULL**.
  - **COUNT** of an empty result is 0.
  - I think that **SUM** on an empty result should be 0 … but it isn't, it's NULL.

- **GROUP BY** does not ignore **NULL**s.
  - The groups that are formed with a **GROUP BY** on attributes A_1, …, A_k may have **NULL** values for one or more (even all) of these attributes.

Examples with NULL

- Suppose r(A,B) is a relation with a single tuple (NULL, NULL).

```
SELECT A, COUNT(B)
FROM r
GROUP BY A;
```

```
SELECT A, COUNT(*)
FROM r
GROUP BY A;
```

```
SELECT A, SUM(B)
FROM r
GROUP BY A;
```

Aggregate Functions – the HAVING Clause

- Find the names and average salaries of all departments whose average salary is greater than 42000.

```
SELECT dept_name, AVG(salary) AS mean_salary
FROM instructor
GROUP BY dept_name
HAVING AVG(salary) > 42000;
```

Note: Predicates in the **HAVING** clause are applied after the formation of groups, whereas predicates in the **WHERE** clause are applied before forming groups.

HAVING Clause

SELECT [**DISTINCT**] $A_1, A_2, \ldots, A_m$, AGGOP(…)
FROM $r_1, r_2, \ldots r_m$
[**WHERE** *P*]
[**GROUP BY** <list of grouping attributes>
[**HAVING** *condition*]]
[**ORDER BY** < list of attributes/expressions [**asc** | **desc**] >] ;

The **HAVING** clause cannot appear without **GROUP BY**.

- **HAVING**: Choose groups based on some aggregate property of the group itself.
  - Think of **HAVING** as like a **WHERE** clause that is applied to groups.
- The attributes and aggregates that may appear in the **SELECT** may also appear in the **HAVING** clause condition.
  - Which attributes and aggregates? Why?

Having Clause Semantics

- Let Result denote an empty multiset of tuples.
- For every tuple t_1 from r_1, t_2 from r_2, …, t_m from r_m
  - If t_1, …, t_n satisfy predicate *P (that is.*, if P evaluates to true), then add the resulting tuple that consists of A_1, A_2, …, A_m components (including attributes of the AGGOP operators) from the t_i into Result.
- Group the tuples in Result according to list of grouping attributes.
  - If **GROUP BY** is omitted, the entire table is regarded as ONE group.
- Apply aggregate operator(s) on the tuples in each group to get a tuple that is put in the Result.
- Apply the condition in the **HAVING** clause to the tuple for each group. Remove tuples that do not satisfy the **HAVING** clause.
- If **DISTINCT** appears in the **SELECT** clause, then remove duplicates from the Result.
- If **ORDER BY** <list of attributes/expressions> appears, then order the tuples in the Result according to the **ORDER BY** clause.
- Return Result.

HAVING Example 1

```
SELECT D.dept_name, SUM(Inst.salary)
FROM department D, instructor Inst
WHERE Inst.dept_name = D.dept_name
      AND D.building = 'Baskin'
GROUP BY D.dept_name
HAVING MIN(Inst.salary) < 50000;
```

Find the department name and total salary for just those departments in Baskin that have at least one instructor whose salary is less than $50,000.

HAVING Example 2

```
SELECT D.dept_name, SUM(Inst.salary)
FROM department D, instructor Inst
WHERE Inst.dept_name = D.dept_name
    AND D.building = 'Baskin'
GROUP BY D.dept_name
HAVING COUNT(DISTINCT Inst.name) >= 4;
```

Find the department name and total salary for just those departments in Baskin that have at least 4 instructors whose names are different.

HAVING Example 3

```
SELECT D.dept_name, SUM(Inst.salary)
FROM department D, instructor Inst
WHERE Inst.dept_name = D.dept_name
    AND D.building = 'Baskin'
GROUP BY D.dept_name
HAVING COUNT(DISTINCT Inst.name) >= 4
    AND MIN(Inst.salary) < 50000;
```

Find the department name and total salary for just those departments in Baskin that have at least 4 instructors whose names are different, and have at least one instructor whose salary is less than $50,000.

Another Example of HAVING Semantics

Find the age of the youngest sailor with age >= 18, for those ratings that have at least 2 sailors whose age is >= 18.

```
SELECT  S.rating, MIN(S.age)
FROM  sailor S
WHERE  S.age >= 18
GROUP BY  S.rating
HAVING COUNT (*) > 1;
```

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
71	Zorba	10	16.0
64	Horatio	7	35.0
92	Frodo	1	28.0
38	Sam	1	30.0
29	Brutus	1	33.0
58	Rusty	10	35.0

Example of Having Semantics: 1

- Take the cross product of all relations in the **FROM** clause.

```
SELECT  S.rating, MIN(S.age)
FROM  sailor S
WHERE  S.age >= 18
GROUP BY  S.rating
HAVING COUNT (*) > 1;
```

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
71	Zorba	10	16.0
64	Horatio	7	35.0
92	Frodo	1	28.0
38	Sam	1	30.0
29	Brutus	1	33.0
58	Rusty	10	35.0

Example of Having Semantics: 2

- Apply the condition in the **WHERE** clause to every tuple.

```
SELECT  S.rating, MIN(S.age)
FROM  sailor S
WHERE  S.age >= 18
GROUP BY  S.rating
HAVING COUNT (*) > 1;
```

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
~~71~~	~~Zorba~~	~~10~~	~~16.0~~
64	Horatio	7	35.0
92	Frodo	1	28.0
38	Sam	1	30.0
29	Brutus	1	33.0
58	Rusty	10	35.0

Example of Having Semantics: 3

- For simplicity, let's ignore the rest of the columns (since they are not needed).

```
SELECT  S.rating, MIN(S.age)
FROM  sailor S
WHERE  S.age >= 18
GROUP BY  S.rating
HAVING COUNT (*) > 1;
```

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
64	Horatio	7	35.0
92	Frodo	1	28.0
38	Sam	1	30.0
29	Brutus	1	33.0
58	Rusty	10	35.0

Example of Having Semantics: 4

- **Sort** the table according to the **GROUP BY** columns.

```
SELECT  S.rating, MIN(S.age)
FROM  sailor S
WHERE  S.age >= 18
GROUP BY  S.rating
HAVING COUNT (*) > 1;
```

rating	age
7	45.0
8	55.5
7	35.0
1	28.0
1	30.0
1	33.0
10	35.0

rating	age
1	28.0
1	30.0
1	33.0
7	35.0
7	45.0
8	55.5
10	35.0

Note: Don't actually have to do a sort to perform GROUP BY

Example of Having Semantics: 5

- Apply condition of **HAVING** clause to each group. Eliminate the groups which do not satisfy the condition in the **HAVING** clause.
- Next, we evaluate the **SELECT** clause.

```
SELECT  S.rating,  MIN(S.age)
FROM  sailor S
WHERE  S.age >= 18
GROUP BY  S.rating
HAVING COUNT (*) > 1;
```

rating	age
1	28.0
1	30.0
1	33.0
7	35.0
7	45.0
~~8~~	~~55.5~~
~~10~~	~~35.0~~

Example of Having Semantics: 6

- Generate one tuple for each group, according to the **SELECT** clause.

```
SELECT  S.rating, MIN(S.age)
FROM  sailor S
WHERE  S.age >= 18
GROUP BY  S.rating
HAVING COUNT (*) > 1;
```

rating	age
1	28.0
7	35.0

What's the Result?

A	B	C	D
a1	b1	1	7
a1	b1	2	8
a2	b1	3	9
a3	b2	4	10
a2	b1	5	11
a1	b1	6	12

```
SELECT A, B, SUM(C), MAX(D)
FROM R
GROUP BY   A, B;
```

What if query asked for B after SUM(C)?

What if query didn't ask for B in the SELECT?

ANY/SOME in HAVING

```
SELECT  S.rating,  MIN(S.age)
FROM  sailor S
WHERE  S.age >= 18
GROUP BY  S.rating
HAVING  COUNT(*) > 1 AND SOME (S.age > 40);
```

rating	age
1	28.0
1	30.0
1	33.0
7	35.0
7	45.0
8	55.5
10	35.0

rating	age
7	35.0

EVERY (not ALL) in HAVING

```
SELECT  S.rating,  MIN(S.age)
FROM  sailor S
WHERE  S.age >= 18
GROUP BY  S.rating
HAVING  COUNT(*) > 1 AND EVERY (S.age <= 40);
```

rating	age
1	28.0
1	30.0
1	33.0
7	35.0
7	45.0
8	55.5
10	35.0

rating	age
1	28.0

Ski Activity Examples

customer

cid	cname	level	type	age
36	Cho	Beginner	snowboard	18
34	Luke	Inter	snowboard	25
87	Ice	Advanced	ski	20
39	Paul	Beginner	ski	33

activity

cid	slopeid	day
36	s3	01/05/13
36	s1	01/06/13
36	s1	01/07/13
87	s2	01/07/13
87	s1	01/07/13
34	s2	01/05/13

slope

slopeid	name	color
s1	Mountain Run	blue
s2	Olympic Lady	black
s3	Magic Carpet	green
s4	KT-22	black

COUNT Examples

- Find the total number of customers.

```
SELECT COUNT(cid)
FROM customer;
```

- Find the total number of days that customers engaged in activities.

```
SELECT COUNT(DISTINCT day)
FROM activity;
```

- Compare to:

```
SELECT COUNT(day)
FROM activity;
```

Alternatively, the last query could have been written as:

```
SELECT COUNT(*)
FROM activity;
```

But only because day can't be NULL

COUNT Examples, with Join

- Find the number of activities done by advanced customers.

  SELECT COUNT(A.cid)

  FROM customer C, activity A

  WHERE A.cid = C.cid

  AND C.level = 'Advanced';

- Find the number of activities done by different advanced customers.

  SELECT COUNT(**DISTINCT** A.cid)

  FROM customer C, activity A

  WHERE A.cid = C.cid

  AND C.level = 'Advanced';

Would these queries have same results with **COUNT**(C.cid) instead of **COUNT**(A.cid)?

What about if queries had **COUNT**(*)/**COUNT**(**DISTINCT** *) instead?

COUNT Examples, with JOIN and GROUP BY

- For each day, find the number of activities that were done by advanced customers on that day.

```
SELECT A.day, COUNT(A.cid)
FROM customer C, activity A
WHERE A.cid = C.cid
    AND C.level = 'Advanced'
GROUP BY A.day;
```

- For each day, find the number of different advanced customers who did at least one activity on that day.

```
SELECT A.day, COUNT(DISTINCT A.cid)
FROM customer C, activity A
WHERE A.cid = C.cid
    AND C.level = 'Advanced'
GROUP BY A.day;
```

COUNT Examples, with JOIN and GROUP BY and HAVING

- For each customer level, find the number of times that customers who are at that level went on a red slope, giving level as well as number of times,
  … but only if the number of times is at least 3.
  - The number of times should appear as redCount in the result.

```
SELECT C.level, COUNT(*) AS redCount
FROM customer C, activity A, slope S
WHERE A.cid = C.cid
    AND A.slopeid = S.slopeid
    AND S.color = 'Red'
GROUP BY C.level
HAVING COUNT(*) >= 3;
```

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries. A **subquery** is a **SELECT-from-WHERE** expression that is nested within another query.
- The nesting can be done in the following SQL query

  SELECT A_1, A_2, ..., A_n
  FROM r_1, r_2, ..., r_m
  WHERE P;

  as follows:

  - **FROM clause:** r_i can be replaced by any valid subquery
  - **WHERE clause:** P can be replaced with an expression of the form:

    B <operation> (subquery)

    where B is an attribute, and <operation> will be described later.
  - **SELECT clause:**

    A_i can be replaced be a subquery that generates a single value.

Set Membership

Set Membership

- Find courses offered in both Fall 2017 and in Spring 2018.

```
SELECT DISTINCT course_id
FROM section
WHERE semester = 'Fall' AND year= 2017 AND
        course_id  IN   ( SELECT course_id
                           FROM section
                           WHERE semester = 'Spring' and year = 2018 );
```

- Find courses offered in Fall 2017 but not in Spring 2018.

```
SELECT DISTINCT course_id
FROM section
WHERE semester = 'Fall' and year= 2017 and
        course_id  NOT IN ( SELECT course_id
                            FROM section
                            WHERE semester = 'Spring' and year = 2018 );
```

Set Membership (Continued)

- Name all instructors whose name is neither 'Mozart', nor 'Einstein'

```
SELECT DISTINCT name
FROM instructor
WHERE name NOT IN ('Mozart', 'Einstein');
```

- Find the total number of (different) students who have taken course sections taught by the instructor whose *ID is* 10101.

```
SELECT COUNT (DISTINCT ID)
FROM takes
WHERE (course_id, sec_id, semester, year) IN
                    ( SELECT course_id, sec_id, semester, year
                      FROM teaches
                      WHERE teaches.ID = 10101 );
```

- Note: The above query can be written in a much simpler manner. The formulation above is simply to illustrate SQL features.

Set Comparison

Set Comparison: SOME

- Find the names of instructors whose salary is greater than the salary of some (at least one) instructor in the Biology department.

```
SELECT DISTINCT Inst1.name
FROM instructor Inst1, instructor Inst2
WHERE Inst1.salary > Inst2.salary
    AND Inst2.dept_name = 'Biology';
```

- Same query using > **SOME** clause

```
SELECT name
FROM instructor
WHERE salary > SOME ( SELECT Inst2.salary
                      FROM instructor Inst2
                      WHERE Inst2.dept_name = 'Biology' );
```

- You can use **ANY** in SQL, instead of **SOME**

Definition of SOME

- x <comp> **SOME** $r \Leftrightarrow \exists\, t \in r$ such that (x <comp> t)
  Where <comp> can be: $<,\ \leq,\ >,\ =,\ <>$

(5 < **SOME** | 0 / 5 / 6 |) = TRUE (read: 5 < some tuple in the relation)

(5 < **SOME** | 0 / 5 |) = FALSE

(5 = **SOME** | 0 / 5 |) = TRUE

(5 <> **SOME** | 0 / 5 |) = TRUE (since 0 ≠ 5)

(= **SOME**) is equivalent to **in**.

However, (<> **SOME**) is not equivalent to **NOT IN**. Why not?

Set Comparison: ALL (not EVERY)

- Find the names of the instructors whose salary is greater than the salary of all the instructors in the Biology department.

```
SELECT name
FROM instructor
WHERE salary > ALL ( SELECT salary
                     FROM instructor
                     WHERE dept name = 'Biology' );
```

Definition of ALL

- x <comp> **ALL** *r* $\Leftrightarrow$ $\forall$ *t* $\in$ *r* (x <comp> *t)*

(5 < **ALL** {0, 5, 6}) is FALSE

(5 < **ALL** {6, 10}) is TRUE

(5 = **ALL** {4, 5}) is FALSE

(5 <> **ALL** {4, 6}) is TRUE (since 5 ≠ 4 and 5 ≠ 6)

"x <> **ALL** Q" is equivalent to "x **NOT IN** Q"

However, "x = **ALL** Q" is not equivalent to "x **IN** Q" Why not?

Test for Empty Relations

- The **EXISTS** construct returns the value TRUE if the argument subquery is non-empty.
- **EXISTS** $r \Leftrightarrow r \neq \emptyset$
- **NOT EXISTS** $r \Leftrightarrow r = \emptyset$

Use of EXISTS

- Here's yet another way to find all courses offered in both the Fall 2017 semester and in the Spring 2018 semester.

```
SELECT Sec1.course_id
FROM section AS Sec1
WHERE Sec1.semester = 'Fall' AND Sec1.year = 2017
   AND EXISTS ( SELECT *
                FROM section AS Sec2
                WHERE Sec2.semester = 'Spring'
                    AND Sec2.year = 2018
                    AND Sec1.course_id = Sec2.course_id );
```

- **Correlation name** – variable *sec1* in the outer query
- **Correlated subquery** – the inner query

Use of NOT EXISTS

- Find all students who have taken all the courses offered in the Biology department.

```
SELECT S.ID, S.name
FROM student AS S
WHERE NOT EXISTS ( ( SELECT C.course_id
                     FROM course AS C
                     WHERE C.dept_name = 'Biology' )
                   EXCEPT
                   ( SELECT T.course_id
                     FROM takes AS T
                     WHERE S.ID = T.ID ) );
```

  - First nested query lists all the courses offered in Biology.
  - Second nested query lists all the courses that the particular student took.

- Note that $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- Note: Cannot write this query using **= ALL** and its variants.

A Subquery Example with Aggregates

For the relation: sailors(sid, sname, rating, age)

- Find the minimum age of sailors for each rating category in which the minimum age of sailors in that category is greater than the average age of all sailors.

SELECT *S.rating*, **MIN**(*S.age*)

FROM *sailor S*

GROUP BY *S.rating*

HAVING MIN(*S.age*) > (**SELECT AVG**(*S2.age*)

FROM *Sailor S2*) ;

Another Subquery with Aggregates

- Find the second minimum age of sailors.

SELECT MIN(*S.age*)
FROM *sailor S*
WHERE *S.age* > (**SELECT MIN**(*S2.age*)
FROM *sailor S2*);

- What happens when there is only one sailor?
- What happens when all sailors have the same age?
- What happens when there are no sailors?
- Can you figure how to find the third minimum age of sailors?

Some Incorrect Examples

Find the second minimum age of sailors.
All the answers below are illegal!

```
SELECT MIN(S.age)
FROM sailor S
WHERE S.age > MIN(S.age);
```

```
SELECT MIN(S.age)
FROM sailor S
WHERE S.age > MIN(S2.age);
```

```
SELECT MIN(S.age)
FROM sailor S
WHERE S.age > ( MIN(S2.age)
                FROM sailor S2 );
```

```
SELECT MIN(S.age)
FROM sailor S
WHERE S.age > MIN( SELECT S2.age
                   FROM sailor S2 );
```

Aggregates can't appear in WHERE clauses, except in legal subqueries, as on previous slide.

Ski Activity Examples

customer

cid	cname	level	type	age
36	Cho	Beginner	snowboard	18
34	Luke	Inter	snowboard	25
87	Ice	Advanced	ski	20
39	Paul	Beginner	ski	33

activity

cid	slopeid	day
36	s3	01/05/13
36	s1	01/06/13
36	s1	01/07/13
87	s2	01/07/13
87	s1	01/07/13
34	s2	01/05/13

slope

slopeid	name	color
s1	Mountain Run	blue
s2	Olympic Lady	black
s3	Magic Carpet	green
s4	KT-22	black

Example 1

- Find the id and name of each customer who did some activity on 01/07/13.

```
SELECT C.cid, C.cname
FROM customer C, activity A
WHERE A.day = DATE '01/07/13'
    AND A.cid = C.cid;
```

Wrong!

```
SELECT C.cid, C.cname
FROM customer C
WHERE C.cid IN ( SELECT A.cid
                 FROM activity A
                 WHERE A.day = DATE '01/07/13' );
```

Example 1 (Equivalent?)

- Find the id and name of each customer who did some activity on the day 01/07/13.
  - Is this equivalent to previous query?

```
SELECT C.cid, C.cname
FROM customer C
WHERE EXISTS ( SELECT *
                 FROM activity A
                 WHERE A.cid = C.cid
                     AND A.day = DATE '01/07/13' );
```

Example 2

- Find the id and name of each customer who did not do any activity on 01/07/13.

```
SELECT C.cid, C.cname
FROM customer C
WHERE C.cid NOT IN ( SELECT A.cid
                     FROM activity A
                     WHERE A.day = DATE '01/07/13' );
```

- Can you figure out a way to write this using **NOT EXISTS**?

Example 3

- Find the id and name of each customer who went on the slope “Olympic Lady” on 01/07/13.

```
SELECT C.cid, C.cname
FROM   customer C
WHERE C.cid IN ( SELECT A.cid
                 FROM activity A
                 WHERE A.day = DATE ‘01/07/13’
                  AND A.slopeid IN ( SELECT S.slopeid
                                     FROM slope S
                                     WHERE S.name = ‘Olympic Lady’)
                  );
```

Example 3 (Equivalent?)

- Find the id and name of each customer who went on the slope “Olympic Lady” on 01/07/13.

```
SELECT DISTINCT C.cid, C.cname
FROM   customer C, activity A, slope S
WHERE C.cid = A.cid
    AND A.day = DATE ‘01/07/13’
    AND S.name = ‘Olympic Lady’
    AND A.slopeid = S.slopeid;
```

Is this equivalent to the previous query?

Example 4

- Determine the colors of all slopes that Cho went on.

SELECT DISTINCT *S.color*
FROM *activity A, slope S*
WHERE *A.slopeid* = *S.slopeid*
AND *A.cid* = (**SELECT** *C.cid*
FROM *customer C*
WHERE *C.cname* = 'Cho');

- Would this be correct, if *cname* **is UNIQUE**?
- Would this be correct, if *cname* **is not UNIQUE**?
- How might you fix this, if *cname* **is not UNIQUE**?

Example 5

- Find the names of all customers who did some activity.

SELECT *C.cname*
FROM *customer C*
WHERE *C.cid* = **ANY** (**SELECT** *A.cid*
FROM *activity A*);

Example 6

- Find the names of all customers who are skiers and whose age is greater than the age of every snowboarder.

SELECT *C.cname*
FROM *customer C*
WHERE *C.type* = 'skier' **AND** *C.age* > **ALL** (**SELECT** *C2.age*
FROM *customer C2*
WHERE *C2.type* = 'snowboard');

What happens if this subquery returns an empty set?

- Find the names of the oldest customers.

  Please try to write this final query yourselves.

MIN, MAX

- Find the name and age of the oldest snowboarders.

  SELECT *C.cname*, **MAX**(*C.age*)
  FROM *customer C*
  WHERE *C.type* = ‘snowboard’;

- WRONG!

- The non-aggregate columns in the **SELECT** clause must come from the attributes in the **GROUP BY** clause.

MIN, MAX with Subquery

- Find the name and age of the oldest snowboarders.

SELECT *C.cname, C.age*
FROM *customer C*
WHERE *C.age* = (**SELECT MAX**(*c2.age*)
FROM *customer C2*
WHERE *c2.type* = 'snowboard');

Will this query execute correctly?

NO!

If not, how would you correct it?

MIN, MAX

- Find the age of the youngest participant for each type of activity that Beginners participate in.

  SELECT *C.type*, **MIN**(*C.age*)
  FROM *customer C*
  WHERE *C.level* =‘ Beginner’;

- Wrong!

- The non-aggregate columns in the **SELECT** clause must come from the attributes in the **GROUP BY** clause.
  - If there is an aggregate in the **SELECT** clause but no **GROUP BY**, then all tuples satisfying the **WHERE** clause are in a single group, and no attributes are in the **GROUP BY** clause, hence … (what?).

MIN, MAX with GROUP BY

- Find the age of the youngest participant for each type of activity that Beginners participate in.

  SELECT *C.type*, **MIN**(*C.age*)
  FROM *customer C*
  WHERE *C.level* = 'Beginner'
  GROUP BY *C.type*;

- Wrong!

- Selects age of youngest Beginner for activity types that Beginners participate in.

MIN, MAX with GROUP BY and EXISTS

- Find the age of the youngest participant for each type of activity that Beginners participate in.

```
SELECT C.type, MIN(C.age)
FROM customer C
WHERE EXISTS ( SELECT *
                 FROM customer C2
                 WHERE c2.level = 'Beginner'
                     AND c2.type = C.type )
GROUP BY C.type;
```

- Right!

- Selects age of youngest participant for activity types that at least one Beginner participates in.

MIN, MAX with GROUP BY and HAVING

- Find the age of the youngest participant for each type of activity that Beginners participate in.

  SELECT *C.type*, **MIN**(*C.age*)
  FROM *customer C*
  GROUP BY *C.type*
  HAVING SOME (*C.level* = 'Beginner');

- Right

- Selects age of youngest participant for activity types that at least one Beginner participates in.

Careful …

- Find the activities of Luke.

```
SELECT *
FROM activity A
WHERE a.cid = (SELECT c.cid
               FROM customer C
               WHERE C.cname='Luke');
```

If there is only one Luke in the customer table, the subquery returns only one cid value. SQL returns that single cid value to be compared with a.cid.

However, if the subquery returns more than one value, a run-time error occurs.

Reminder: Use of ALL

- Find the names of all customers whose age is greater than the age of every snowboarder.

```
SELECT C.name
FROM customer C
WHERE C.age > ALL  (SELECT C2.age
                    FROM customer C2
                    WHERE C2.type = 'snowboard');
```

What happens if there are no snowboarders?

```
SELECT C.name
FROM customer C
WHERE C.age > (SELECT MAX(C2.age)
               FROM customer C2
               WHERE C2.type = 'snowboard');
```

What happens if there are no snowboarders?

Reminder: Use of ANY/SOME (Synonyms)

- Find the names of all customers whose age is greater than the age of some snowboarder.

SELECT *C.name*
FROM *customer* C
WHERE *C.age* > **SOME** (**SELECT** *C2.age*
FROM *customer* C2
WHERE *C2.type* = 'snowboard');

What happens if there are no snowboarders?

SELECT *c.name*
FROM *customer* C
WHERE *C.age* > (**SELECT MIN**(*C2.age*)
FROM *customer* C2
WHERE *C2.type* = 'snowboard');

What happens if there are no snowboarders?

GROUP BY: Relaxing a Restriction

- For each cid, give cid and the days on which customer did an activity, with COUNT of number of activities done on that day.

  SELECT *C.cid*, *A.day*, **COUNT**(*)
  FROM *customer* C, *activity* A
  WHERE *A.cid* = *C.cid*
  GROUP BY *C.cid*, *A.day*;

- For each cid, give cid, level and the days on which customer did an activity , with COUNT of number of activities done on that day.

  SELECT *C.cid*, *C.level*, *A.day*, **COUNT**(*)
  FROM *customer* C, *activity* A
  WHERE *A.cid* = *C.cid*
  GROUP BY *C.cid*, *A.day*;

  - Is this legal SQL?
  - We'll accept this on tests and homeworks, if you do it right.

Even though level isn't a **GROUP BY** attribute, this is okay in most SQL databases!

Why? cid is the entire Primary Key of the customer table, so level is not ambiguous.

Subqueries in the FROM Clause

Subqueries in the FROM Clause

- SQL allows a subquery expression to appear in the **FROM** clause.
- Find the dept_name and average instructor salary for departments in which the average salary is greater than $42,000.

```
SELECT dept_name, mean_salary
FROM ( SELECT dept_name, AVG (salary) AS mean_salary
          FROM instructor
          GROUP BY dept_name )
WHERE mean_salary > 42000;
```

- Note that we used the **WHERE** clause, not the **HAVING** clause!
- Another similar way to write the same query:

```
SELECT M.dept_name, M.mean_salary
FROM (  SELECT Inst.dept_name, AVG(Inst.salary)
           FROM instructor Inst
           GROUP BY Inst.dept_name  )
       AS M (dept_name, mean_salary)
WHERE M.mean_salary > 42000;
```

Subqueries in the From Clause (cont.)

- Find the dept_name, average instructor salary, building and budget for departments in which the average salary is greater than $42,000.

```
SELECT M.dept_name, M.mean_salary,
        D.building, D.budget
FROM department D,
    ( SELECT Inst.dept_name, AVG(Inst.salary)
      FROM instructor Inst
      GROUP BY Inst.dept_name )
    AS M(dept_name, mean_salary),
WHERE d.dept_name = M.dept_name
  AND M.mean_salary > 42000;
```

Scalar Subquery

- A scalar subquery is one which is used where a single value is expected
- List all departments, along with the number of instructors in each department

```
SELECT dept_name,
        ( SELECT COUNT(*)
          FROM instructor
          WHERE department.dept_name = instructor.dept_name )
               AS num_instructors
FROM department;
```

- There will be a runtime error if a subquery that's expected to be a scalar subquery returns more than one result tuple.
  - Can the above subquery ever return more than one result tuple?

Scalar Subquery (cont.)

Suppose that you want to find the ID of all the students taking a Comp. Sci. course called 'Database Systems'. You might write:

```
SELECT T.ID
FROM takes T
WHERE T.course_ID =
                ( SELECT C.course_ID
                  FROM course C
                  WHERE C.dept_name = 'Comp. Sci.'
                      AND C.title = 'Database Systems' );
```

- This query will return a runtime error if there is more than one Comp. Sci. course called 'Database Systems'!
- How could you fix this query so that it correctly finds the ID of all students taking a Comp. Sci. course called 'Database Systems'?

Practice Homework

- Beers(name,manufacturer)
- Bars(name,address,license)
- Sells(bar,beer,price)
- Drinkers(name,address,phone)
- Likes(drinker,beer)
- Frequents(drinker,bar)
- Friends(drinker1, drinker2)

1. Find all beers liked by two or more drinkers.
2. Find all beers liked by three or more drinkers.
3. Find all beers liked by friends of Anna.
4. Find all bars that sell a beer that is cheaper than all beers sold by the bar '99 Bottles'.

Modification of the Database

- Deletion of tuples from a given relation
- Insertion of new tuples into a given relation
- Updating of values in some tuples in a given relation

Deletion

- Delete all instructors

  DELETE FROM *instructor;*

- Delete all instructors from the Finance department

  DELETE FROM *instructor*
  WHERE *dept_name* = 'Finance';

- Delete tuples in the *instructor* relation if those instructors are associated with a department located in the Watson building.

  DELETE FROM *instructor*
  WHERE *dept_name* **IN** (**SELECT** *D.dept_name*
  FROM *department D*
  WHERE *D.building* = 'Watson');

Deletion (Cont.)

- Delete all instructors whose salary is less than the average salary of instructors

  DELETE FROM *instructor*
  WHERE *salary* < (**SELECT AVG**(*salary*)
  FROM *instructor*);

  - Problem: As we delete tuples from deposit, the average salary changes!
    - So the deleted instructors could depend on the order in which instructors are inspected! That would be poor semantics.
  - Solution used in SQL:
    1. First, compute **AVG** (salary) and determine the tuples to delete.
    2. Next, delete those tuples, without recomputing the average.

Insertion

- Add a new tuple to *course*.

  INSERT INTO *course*
  VALUES ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

  … or equivalently:

  INSERT INTO *course* (*course_id*, *title*, *dept_name*, *credits*)
  VALUES ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

  … or equivalently:

  INSERT INTO *course* (*title* , *credits*, *dept_name*, *course_id*)
  VALUES ('Database Systems',4, 'Comp. Sci.' , 'CS-437');

- Add a new tuple to *student,* with *tot_creds* set to **NULL**.

  INSERT INTO *student*
  VALUES ('3003', 'Green', 'Finance', **NULL**);

  … or equivalently:

  INSERT INTO *student*
  VALUES ('3003', 'Green', 'Finance');

Insertion, DEFAULT and NULL

CREATE TABLE *instructor* (
ID **char**(5),
name **VARCHAR**(20)**,**
dept_name **VARCHAR**(20) **NOT NULL DEFAULT** 'Comp. Sci.',
salary **NUMERIC**(8,2));

- **INSERT INTO** *instructor*(*ID*, *salary*) **VALUES (**'10211', 80000.00);

- If no value is entered for an attribute and there is a default value, then the value of the attribute will be the default value.
- If no value is entered for an attribute, and no default value is declared explicitly, and the value of the attribute can be **NULL**, then the value of the attribute will default to **NULL**.
  - **NOT NULL** constraint prevents a column from having **NULL** values.
  - Attributes that don't have **NOT NULL** specified may be **NULL**.
  - … but remember that attributes in the PRIMARY KEY cannot be **NULL**.
- If no value is entered for an attribute and no default value is declared explicitly, and the value of the attribute cannot be **NULL …**
  - **…** then there's an error.

Insertion (Cont.)

- Make each student in the Music department who has earned more than 144 credit hours become an instructor in the Music department, with a salary of $18,000.

```
INSERT INTO instructor
    SELECT ID, name, dept_name, 18000
    FROM   student
    WHERE  dept_name = 'Music' AND total_cred > 144;
```

- The **SELECT FROM WHERE** statement is evaluated fully before any of its results are inserted into the relation.

  Otherwise, statements such as:

```
INSERT INTO table1
    SELECT * FROM table1;
```

  wouldn't make sense.

Updates

- Give a 5% salary raise to all instructors.

```
UPDATE instructor
    SET salary = salary * 1.05;
```

- Give a 5% salary raise to those instructors who earn less than 70000.

```
UPDATE instructor
    SET salary = salary * 1.05
    WHERE salary < 70000;
```

- Give a 5% salary raise to instructors whose salary is less than average.

```
UPDATE instructor
SET salary = salary * 1.05
WHERE salary < ( SELECT AVG(salary)
                 FROM instructor );
```

Updates (Cont.)

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others by a 5%
  - Write two **UPDATE** statements:

```
UPDATE instructor
    SET salary = salary * 1.03
    WHERE salary > 100000;

UPDATE instructor
    SET salary = salary * 1.05
    WHERE salary <= 100000;
```

  - The order is important!
  - Can be done more easily, clearly and correctly using the **CASE** construct (next slide).

Case Statement for Conditional Updates

- Same query as before but with the **CASE** construct:

```
UPDATE instructor
    SET salary = CASE
                    WHEN salary <= 100000 THEN salary * 1.05
                    ELSE salary * 1.03
                 END;
```

- General format of the **CASE** construct is:

```
CASE
  WHEN pred1 THEN result1
  WHEN pred2 THEN result2
       …
  WHEN predk THEN resultk
  ELSE result0
END
```

Updates with Scalar Subqueries

- Recompute the total credits taken for each student, and update the student's *tot_cred* value accordingly .

```
UPDATE student S
SET tot_cred = ( SELECT SUM(course.credits)
                 FROM takes, course
                 WHERE takes.course_id = course.course_id
                     AND S.ID= takes.ID
                     AND takes.grade <> 'F'
                     AND takes.grade IS NOT NULL );
```

- But this sets *tot_creds* to **NULL** for students who have not taken any courses! How can we set *tot_creds* correctly to 0?

- Instead of **SELECT SUM**(*credits*) in the above query, write:

```
SELECT CASE
    WHEN SUM(credits) IS NOT NULL
        THEN SUM(credits)
    ELSE 0
END
```

We'll discuss another way to do this later in the term.