Administrative Notes – February 16, 2023

- Feb 17: Assignment 3 due
  - Office hours and tutorials are great places to get help
- Feb 17: Assignment 4 will be released
- Feb 20 – 24: Reading Break! (yay!)
  - No lectures, tutorials, or office hours during this week

Review: GROUP BY and HAVING (cont)

```
SELECT [DISTINCT]  target-list
FROM relation-list
WHERE qualification
GROUP BY grouping-list
HAVING group-qualification
ORDER BY target-list
```

- The *target-list* contains
  (i) attribute names
  (ii) terms with aggregate operations (e.g., `MIN(S.age)`).
- Attributes in (i) must also be in *grouping-list*.
  - each answer tuple corresponds to a *group*,
  - *group* = a set of tuples with same value for all attributes in *grouping-list*
  - selected attributes must have a single value per group.
- Attributes in *group-qualification* are either in *grouping-list* or are arguments to an aggregate operator.

Grouping Examples (cont')

For each standing, find the number of students who took a class with "System" in the title

```
SELECT s.standing, COUNT(DISTINCT s.snum) AS scount
FROM Student S, enrolled E
WHERE S.snum = E.snum and E.cname like '%System%'
GROUP BY s.standing
```

What if we do the following:
a) remove *E.cname like '%System%'* from the WHERE clause, and then

b) add a HAVING clause with the dropped condition?

```
SELECT  s.standing,  COUNT(DISTINCT s.snum) AS scount
FROM       Student S, enrolled E
WHERE    S.snum = E.snum
GROUP BY  s.standing
HAVING  E.cname like '%System%'
```

E.Cname not in groupby Error!

Clicker Question: Having

Suppose we have a relation with schema R(A, B, C, D, E). If we issue a query of the form:

```
SELECT ...
FROM R
WHERE ...
GROUP BY B, E
HAVING ???
```

Identify, in the list below, the term that CANNOT appear in the HAVING clause (where the ??? is).

A. A (unaggregated)
B. B (unaggregated)
C. Count(B)
D. All can appear
E. None can appear

Clicker Question: Having

Suppose we have a relation with schema R(A, B, C, D, E). If we issue a query of the form:

```
SELECT ...
FROM R
WHERE ...
GROUP BY B, E
HAVING ???
```

Identify, in the list below, the term that CANNOT appear in the HAVING clause (where the ??? is).

A. A (unaggregated)
B. B (unaggregated)
C. Count(B)
D. All can appear
E. None can appear

Any aggregated term can appear in HAVING clause. An attribute not in the GROUP-BY list cannot be unaggregated in the HAVING clause. Thus, B or E may appear unaggregated, and all five attributes can appear in an aggregation. However, A, C, or D cannot appear alone.

Grouping Examples (cont')

Find the age of the youngest student with age > 18, for each major with at least 2 students(of age > 18).

Student(snum,sname,major,standing,age)
Class(name,meets_at,room,fid)
Enrolled(snum,cname)
Faculty(fid,fname,deptid)

Grouping Examples (cont')

Find the age of the youngest student with age > 18, for each major with at least 2 students(of age > 18).

```
SELECT S.major, MIN(S.age)
FROM   Student S
WHERE  S.Age > 18
GROUP BY S.major
HAVING COUNT(*) > 1
```

Student(snum,sname,major,standing,age)
Class(name,meets_at,room,fid)
Enrolled(snum,cname)
Faculty(fid,fname,deptid)

Grouping Examples (cont')

Find the age of the youngest student with age > 18, for each major for which the average age of the students who are >18 is higher than the average age of all students across all majors.

Student(snum,sname,major,standing,age)
Class(name,meets_at,room,fid)
Enrolled(snum,cname)
Faculty(fid,fname,deptid)

Grouping Examples (cont')

Find the age of the youngest student with age > 18, for each major for which the average age of the students who are >18 is higher than the average age of all students across all majors.

```
SELECT S.major, MIN(S.age), avg(S.age)
FROM  Student S
WHERE  S.age > 18
GROUP BY  S.major
HAVING avg(S.age) > (SELECT avg(age)
                     FROM Student)
```

Note: avg(S.age) is included as a piece of information for your reference. The question doesn't indicate that you need to include this in the answer.

Grouping Examples (cont')

Find the age of the youngest student with age > 18, for each major for which the average age of the students who are >18 is higher than the average age of all students across all majors.

```
SELECT S.major, MIN(S.age), avg(S.age)
FROM  Student S, Student S2
WHERE  S.age > 18 AND S.snum = S2.snum
GROUP BY  S.major
HAVING avg(S.age) > avg(S2.age)
```

Issue: avg(S.age) would have the same value as avg(S2.age)

Grouping Examples (cont')

Student table (some attributes omitted)

snum	major	age
1	CS	18
2	Music	20
3	Music	19
4	English	17
5	Business	21

Joining two instances of student and removing S.age <= 18

S.snum	S.major	S.age	S2.snum	S2.major	S2.age
~~1~~	~~CS~~	~~18~~	~~1~~	~~CS~~	~~18~~
2	Music	20	2	Music	20
3	Music	19	3	Music	19
~~4~~	~~English~~	~~17~~	~~4~~	~~English~~	~~17~~
5	Business	21	5	Business	21

Grouping Examples (cont')

Grouping by major (each group shown separately)

S.snum	S.major	S.age	S2.snum	S2.major	S2.age
2	Music	20	2	Music	20
3	Music	19	3	Music	19

S.snum	S.major	S.age	S2.snum	S2.major	S2.age
5	Business	21	5	Business	21

Taking the average age of S.age would be the same as taking the average age of S2.age.

Grouping Examples (cont')

Find the age of the youngest student with age > 18, for each major with at least 2 students (of any age).

Grouping Examples (cont')

Find the age of the youngest student with age > 18, for each major with at least 2 students(of any age)

```
SELECT S.major, MIN(S.age)
FROM Student S
WHERE S.age > 18
GROUP BY S.major
HAVING 1 < (SELECT  COUNT(*)
            FROM  Student S2
            WHERE  S.major=S2.major)
```

- Subqueries in the HAVING clause can be correlated with fields from the outer query.

Grouping Examples (cont')

Find those majors for which their average age is the minimum over all majors

```
SELECT major, avg(age)
FROM   student S
GROUP BY major
HAVING min(avg(age))
```

- WRONG, cannot use nested aggregation
  - One solution would be to use subquery in the FROM Clause

```
SELECT  Temp.major, Temp.average
FROM (SELECT  S.major, AVG(S.age) as average
      FROM  Student S
      GROUP BY S.major) AS Temp
WHERE Temp.average in (SELECT  MIN(Temp.average)
                       FROM  Temp)
```

A bit ugly

Grouping Examples (cont')

Find those majors for which their average age is the minimum over all majors

```
SELECT major, avg(age)
FROM   student S
GROUP BY major
HAVING min(avg(age))
```

- WRONG, cannot use nested aggregation
  - One solution would be to use subquery in the FROM Clause

```
SELECT major, avg(age)
FROM   student S
GROUP BY major
HAVING AVG(age) <= ALL (SELECT  AVG(S.age)
                        FROM   Student S
                        GROUP BY S.major)
```

Easiest method would be to use Views.

In-Class Exercise (SQL 4)

- Canvas → Modules → In Class Exercises
- You can work on it with other people around you. If you work with others, you must **write their names on your submission to acknowledge the collaboration**.
  - Everyone must submit to Canvas
- Reminder: no late submissions accepted

Clicker Question

I am ready to cover the in-class exercise.

A. Yes
B. No
C. I need two more minutes
D. I need five more minutes

What are views

- Relations that are defined with a `CREATE TABLE` statement exist in the physical layer
  - Do not change unless explicitly told so

- Virtual views do not physically exist, they are defined by expression over the tables.
  - Can be queries (most of the time) as if they were tables.

Why use views?

- Hide some data from users
- Make some queries easier
- Modularity of database
- When not specified exactly based on tables.

Example: UBC has one table for students. Should the CS Department be able to update CS students info? Yes.

Biology students? NO

Create a view for CS to only be able to update CS students

Defining and using Views

- `Create View <view name><attributes in view> As <view definition>`
  - View definition is defined in SQL
  - From now on we can use the view almost as if it is just a normal table
- View V (R_1,…R_n)
- query Q involving V
  - Conceptually
    - V (R_1,…R_n) is used to evaluate Q
  - In reality
    - The evaluation is performed over R_1,…R_n

Defining and using Views

Suppose you had tables

Course(Course#,title,dept)

Enrolled(Course#,sid,mark)

```
CREATE  VIEW  CourseWithFails(dept,
course#, mark) AS
    SELECT C.dept, C.course#, mark
    FROM Course C, Enrolled E
    WHERE C.course# = E.course# AND
    mark<50
```

This view gives the dept, course#, and marks for those courses where someone failed.

Views and Security

- Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s).
- Given CourseWithFails, but not Course or Enrolled, we can find the course in which some students failed, but we can't find the students who failed.

```
Course(Course#,title,dept)
Enrolled(Course#,sid,mark)
VIEW  CourseWithFails(dept, course#, mark)
```

View Updates

- View updates must occur at the base tables.
  - Ambiguous
  - Difficult

CourseWithFails(dept, course#, mark)

Course(Course#,title,dept)
Enrolled(Course#,sid,mark)

- DBMS's restrict view updates only to some simple views on single tables (called updatable views)

View Deletes

- `DROP VIEW <view name>`
  - Dropping a view does not affect any tuples of the in the underlying relation.

- How to handle DROP TABLE if there's a view on the table?
- DROP TABLE command has options to prevent a table from being dropped if views are defined on it:
  - DROP TABLE Student RESTRICT
    - drops the table, unless there is a view on it
  - DROP TABLE Student CASCADE
    - drops the table, and recursively drops any view referencing it

The Beauty of Views

Find those majors for which their average age is the minimum over all majors.

With views:

```
CREATE VIEW Temp(major, average) as
   SELECT S.major, AVG(S.age) AS average
   FROM Student S
   GROUP BY  S.major;

SELECT major, average
FROM Temp
WHERE average = (SELECT MIN(average) FROM Temp)
```

Without views:

A bit ugly

```
SELECT Temp.major, Temp.average
FROM (SELECT  S.major, AVG(S.age) as average
      FROM  Student S
      GROUP BY S.major) AS Temp
WHERE Temp.average in (SELECT  MIN(Temp.average)
                       FROM  Temp)
```

Clicker Question: Views

Suppose relation `R(a,b,c)`:

Define the view *V* by:

```
CREATE VIEW V AS
     SELECT a+b AS d, c
     FROM R;
```

What is the result of the query:

```
SELECT d, SUM(c)
FROM V
GROUP BY d
HAVING COUNT(*) <> 1;
```

R

a	b	c
1	1	3
1	2	3
2	1	4
2	3	5
2	4	1
3	2	4
3	3	6

Identify, from the list below, a tuple in the result of the query:

A. (2,3)

B. (3,12)

C. (5,9)

D. All are correct

E. None are correct

Clicker Question: Views

Suppose relation `R(a,b,c)`:

Define the view *V* by:

```
CREATE VIEW V AS
     SELECT a+b AS d, c
     FROM R;
```

What is the result of the query:

```
SELECT d, SUM(c)
FROM V
GROUP BY d
HAVING COUNT(*) <> 1;
```

R

a	b	c
1	1	3
1	2	3
2	1	4
2	3	5
2	4	1
3	2	4
3	3	6

V

d	c
2	3
3	3
3	4
5	5
6	1
5	4
6	6

d	sum(c)
3	7
5	9
6	7

Identify, from the list below, a tuple in the result of the query:

A. (2,3)

B. (3,12)

C. (5,9)

D. All are correct

E. None are correct

Clickerview.sql

Null Values

- Tuples may have a null value, denoted by *null*, for some of their attributes
- Value *null* signifies an unknown value or that a value does not exist.
- The predicate **IS NULL** (**IS NOT NULL**) can be used to check for null values.
  - E.g., *Find all student names whose age is not known.*

```
SELECT name
FROM Student
WHERE age IS NULL
```

- The result of any arithmetic expression involving *null* is *null*
  - E.g., 5 + *null* returns *null.*

Null Values and Three Valued Logic

- null requires a 3-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*
- Any comparison with *null* returns *unknown*
  - *E.g. 5 < null or null <> null or null = null*
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*
- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes.

```
SELECT count(*)          SELECT count(fid)
FROM class               FROM class
```

Clicker Question: Null

```
SELECT COUNT(*),
       COUNT(Runs)
FROM Scores
WHERE Team = 'Carp'
```

Which of the following is in the result:

A. (1,0)

B. (2,0)

C. (1,NULL)

D. All of the above

E. None of the above

Scores

Team	Day	Opponent	Runs
Dragons	Sun	Swallows	4
Tigers	Sun	Bay Stars	9
Carp	Sun	NULL	NULL
Swallows	Sun	Dragons	7
Bay Stars	Sun	Tigers	2
Giants	Sun	NULL	NULL
Dragons	Mon	Carp	NULL
Tigers	Mon	NULL	NULL
Carp	Mon	Dragons	NULL
Swallows	Mon	Giants	0
Bay Stars	Mon	NULL	NULL
Giants	Mon	Swallows	5

Clicker Question: Null

Start clickernull.sql

```
SELECT COUNT(*),
       COUNT(Runs)
FROM Scores
WHERE Team = 'Carp'
```

Which of the following is in the result:

A. (1,0)

B. (2,0)

C. (1,NULL)

D. All of the above

E. None of the above

Scores

Team	Day	Opponent	Runs
Dragons	Sun	Swallows	4
Tigers	Sun	Bay Stars	9
Carp	Sun	NULL	NULL
Swallows	Sun	Dragons	7
Bay Stars	Sun	Tigers	2
Giants	Sun	NULL	NULL
Dragons	Mon	Carp	NULL
Tigers	Mon	NULL	NULL
Carp	Mon	Dragons	NULL
Swallows	Mon	Giants	0
Bay Stars	Mon	NULL	NULL
Giants	Mon	Swallows	5

Natural Join

- The SQL NATURAL JOIN is a type of EQUI JOIN and is structured in such a way that, columns with same name of associate tables will appear once only.
- Natural Join : Guidelines
  - The associated tables have one or more pairs of identically named columns.
  - The columns must be the same data type.
  - Don’t use ON clause in a natural join.

```
SELECT *
FROM student s natural join enrolled e
```

- Natural join of tables with no pairs of identically named columns will return the cross product of the two tables.

```
SELECT *
FROM student s natural join class c
```

More fun with joins

- What happens if I execute query:

```
SELECT *
FROM student s, enrolled e
WHERE s.snum = e.snum
```

- To get *all* students, you need an *outer join*
- There are several special joins declared in the *FROM* clause:
  - Inner join – default: only include matches
  - Left outer join – include all tuples from left hand relation
  - Right outer join – include all tuples from right hand relation
  - Full outer join – include all tuples from both relations

Example:

```
SELECT  *
FROM  Student S  NATURAL LEFT OUTER JOIN
Enrolled E
```

More fun with joins examples

R

A	B
1	2
3	3

S

B	C
2	4
4	6

Natural Inner Join

A	B	C
1	2	4

Natural Left outer Join

A	B	C
1	2	4
3	3	Null

Natural Right outer Join

A	B	C
1	2	4
Null	4	6

Natural outer Join

A	B	C
1	2	4
3	3	Null
Null	4	6

Outer join (without the Natural) will use the key word ON for specifying the condition of the join.

Outer join not implemented in MYSQL
Outer join is implemented in Oracle