CPSC 304 – March 13, 2018 Administrative Notes

- Midterm #2 is handed back – see Piazza
  - In general, you all did really well!
  - Regrade requests due in the next week – see a member of the instructional staff, then post on Piazza
- Reminder: Tutorial this week is ungraded – either JDBC or PHP – do what makes the most sense for your project
- Reminder: Tutorial is due this week

Now where were we…

- We'd been discussing SQL, particularly
  SELECT
  FROM
  WHERE
- We'd also said that you can do ordering with ORDER BY
- But what if we want something more complex than just one SELECT FROM WHERE clause?

Set Operations

- **union, intersect,** and **except** correspond to the relational algebra operations $\cup$, $\cap$, $-$.
- Each automatically eliminates duplicates;
  To retain all duplicates use the corresponding multiset versions:

  union all, intersect all and **except all.**
- Suppose a tuple occurs *m* times in *r* and *n* times in *s,* then, it occurs:
  - *m* + *n* times in *r* **union all** *s*
  - min(*m,n)* times in *r* **intersect all** *s*
  - max(0, *m* – *n)* times in *r* **except all** *s*

Find IDs of MovieStars who've been in a movie in 1944 *or* 1974

Find IDs of MovieStars who've been in a movie in 1944 *or* 1974

- **UNION:** Can union any two *union-compatible* sets of tuples (i.e., the result of SQL queries).

```
SELECT  StarID
FROM  Movie M, StarsIn S
WHERE  M.MovieID=S.MovieID AND
( year = 1944 OR year = 1974)
```

- The two queries though quite similar return different results, why?
  - Use UNION ALL to get the same answer

```
SELECT  StarID
FROM     Movie M, StarsIn S
WHERE   M.MovieID = S.MovieID AND
year = 1944
UNION
SELECT  StarID
FROM     Movie M, StarsIn S
WHERE   M.MovieID = S.MovieID AND
year = 1974
```

Set Operations: Intersect

Example: Find IDs of stars who have been in a movie in 1944 *and* 1974.

- **INTERSECT:** Can be used to compute the intersection of any two *union-compatible* sets of tuples.
- In SQL/92, but some systems don't support it.

Set Operations: Intersect

Example: Find IDs of stars who have been in a movie in 1944 *and* 1974.

- **INTERSECT:** Can be used to compute the intersection of any two *union-compatible* sets of tuples.
- In SQL/92, but some systems don't support it.

```
SELECT  StarID
FROM    Movie M, StarsIn S
WHERE   M.MovieID = S.MovieID AND
year = 1944
INTERSECT
SELECT  StarID
FROM    Movie M, StarsIn S
WHERE   M.MovieID = S.MovieID AND
year = 1974
```

Oracle does
MYSQL doesn't

Rewriting INTERSECT with Joins

- Example: Find IDs of stars who have been in a movie in 1944 *and* 1974 without using **INTERSECT.**

Rewriting INTERSECT with Joins

- Example: Find IDs of stars who have been in a movie in 1944 *and* 1974 without using **INTERSECT.**

```
SELECT  distinct S1.StarID
FROM    Movie M1, StarsIn S1,
        Movie M2, StarsIn S2
WHERE
        M1.MovieID = S1.MovieID AND M1.year = 1944 AND
        M2.MovieID = S2.MovieID AND M2.year = 1974 AND
        S2.StarID = S1.StarID
```

Set Operations: EXCEPT

- Find the sids of all students who took Operating System Design but did not take Database Systems

Set Operations: EXCEPT

- Find the sids of all students who took Operating System Design but did not take Database Systems

```
SELECT snum
FROM enrolled e
WHERE cname = 'Operating System Design'
EXCEPT  ← Oracle uses MINUS rather than EXCEPT
SELECT snum
FROM enrolled e
WHERE cname = 'Database Systems'
```

Can we do it in a different way?
(We'll come back to this)

Motivating Example for Nested Queries

Find ids and names of female stars who have been in movie with ID 28:

```
SELECT  M.StarID, name
FROM    MovieStar M, StarsIn S
WHERE  M.StarID = S.starID AND S.MovieID = 28
AND gender = 'female';
```

- *Find ids and names of female stars who have not been in movie w/ ID 28 w/o using EXCEPT/MINUS:*

Would the following be correct?

```
SELECT  M.StarID, name
FROM    MovieStar M, StarsIn S
WHERE  M.StarID = S.starID AND S.MovieID <> 28
and gender = 'female';
```

Nested Queries

- A very powerful feature of SQL:

```
SELECT  A1,A2,…,An
FROM    R1,R2, …,Rm
WHERE   condition
```

- A nested query is a query that has another query embedded with it.
  - A **SELECT, FROM, WHERE, or HAVING** clause can itself contain an SQL query!
  - Being part of the WHERE clause is the most common

Nested Queries (IN/Not IN)

Find ids and names of stars who have been in movie with ID 28:

Nested Queries (IN/Not IN)

Find ids and names of female stars who have been in movie with ID 28:

```
SELECT  M.StarID, M.Name
FROM    MovieStar M
WHERE   M.Gender =  'female' AND
        M.StarID IN  (SELECT  S.StarID
                      FROM  StarsIn S
                      WHERE  MovieID=28)
```

There's also NOT IN

- To find stars who have *not* been in movie 28, use **NOT IN**.
- To understand nested query semantics, think of a *nested loops* evaluation:
  - *For each MovieStar tuple, check the qualification by computing the subquery.*

Nested Queries (IN/Not IN)

Find ids and names of female stars who have been in movie with ID 28:

```
SELECT  M.StarID, M.Name
FROM    MovieStar M
WHERE   M.Gender =  'female' AND
        M.StarID IN  (SELECT  S.StarID
                      FROM  StarsIn S
                      WHERE  MovieID=28)
```

- In this example in inner query does not depend on the outer query so it could be computed just once.
- Think of this as a function that has no parameters.

```
SELECT  S.StarID
FROM  StarsIn S
WHERE  MovieID=28
```

StarID
1026
1027

```
SELECT  M.StarID, M.Name
FROM    MovieStar M
WHERE   M.Gender = 'female' AND
        M.StarID IN
        (1026,1027)
```

Nested Queries with Correlation Same idea, subtle difference

Find names of stars who have been in movie w/ ID 28:

```
SELECT  M.Name
FROM    MovieStar M
WHERE  EXISTS  (SELECT  *
                FROM  StarsIn S
                WHERE  MovieID=28 AND S.StarID = M.StarID)
```

- **EXISTS**: *returns true if the set is not empty*.
- **UNIQUE**: *returns true if there are no duplicates*.
- Illustrates why, in general, subquery must be re-computed for each StarsIn tuple.

Rewriting EXCEPT Queries Using In

- Using nested queries, find the sids of all students who took Operating System Design but did not take Database Systems

Rewriting EXCEPT Queries Using In

- Using nested queries, find the sids of all students who took Operating System Design but did not take Database Systems

```
SELECT snum
FROM enrolled
WHERE cname = 'Operating System Design' and snum not in
        (SELECT snum
         FROM enrolled
         WHERE cname = 'Database Systems')
```

Rewriting INTERSECT Queries Using IN

Find IDs of stars who have been in movies in 1944 and 1974

Rewriting INTERSECT Queries Using IN

Find IDs of stars who have been in movies in 1944 and 1974

```
SELECT S.StarID
FROM    Movie M, StarsIn S
WHERE  M.MovieID = S.MovieID AND M.year = 1944 AND
 S.StarID IN (SELECT  S2.StarID
                FROM Movie M2, StarsIn S2
                WHERE   M2.MovieID = S2.MovieID AND M2.year = 1974)
```

The subquery finds stars who have been in movies in 1974

SQL EXISTS Condition

- The SQL EXISTS condition is used in combination with a subquery and is considered to be met, if the subquery returns at least one row. It can be used in a SELECT, INSERT, UPDATE, or DELETE statement.
- We can also use NOT EXISTS

SQL EXISTS Condition

- Using the EXISTS/ NOT EXISTS operations and correlated queries, find the name and age of the oldest student(s)

SQL EXISTS Condition

- Using the EXISTS/ NOT EXISTS operations and correlated queries, find the name and age of the oldest student(s)

```
SELECT sname, age
FROM student s2
WHERE NOT EXISTS(SELECT *
                 FROM student s1
                 WHERE s1.age >s2.age)
```

More on Set-Comparison Operators

- We've already seen **IN and EXISTS**. Can also use **NOT IN**, **NOT EXISTS**.
- Also available: **op ANY**, **op ALL**, where op is one of: **>, <, =, <=, >=, <>**
- Find movies made after "Fargo"

More on Set-Comparison Operators

- We've already seen **IN and EXISTS**. Can also use **NOT IN**, **NOT EXISTS**.
- Also available: **op ANY**, **op ALL**, where op is one of: **>, <, =, <=, >=, <>**
- Find movies made after "Fargo"

```
SELECT  *
FROM    Movie
WHERE   year > ANY  (SELECT  year
                     FROM    Movie
                     WHERE   Title ='Fargo')
```

Just returning one column

If we have multiple movies names Fargo then we can use ALL instead of ANY

Clicker nested question

Determine the result of:

```
SELECT Team, Day
FROM Scores S1
WHERE Runs <= ALL
    (SELECT Runs
    FROM Scores S2
    WHERE S1.Day = S2.Day )
```

Which of the following is in the result:

A. (Carp, Sun)
B. (Bay Stars, Sun)
C. (Swallows, Mon)
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	Giants	2
Swallows	Sun	Dragons	7
Bay Stars	Sun	Tigers	2
Giants	Sun	Carp	4
Dragons	Mon	Carp	6
Tigers	Mon	Bay Stars	5
Carp	Mon	Dragons	3
Swallows	Mon	Giants	0
Bay Stars	Mon	Tigers	7
Giants	Mon	Swallows	5

Clicker nested question

Clickernested.sql

Determine the result of:

```
SELECT Team, Day
FROM Scores S1
WHERE Runs <= ALL
    (SELECT Runs
    FROM Scores S2
    WHERE S1.Day = S2.Day )
```

Which of the following is in the result:

A. (Carp, Sun)
B. (Bay Stars, Sun)
C. (Swallows, Mon)
D. All of the above
E. None of the above

Correct

Scores:

Team	Day	Opponent	Runs
Dragons	Sun	Swallows	4
Tigers	Sun	Bay Stars	9
Carp	Sun	Giants	2
Swallows	Sun	Dragons	7
Bay Stars	Sun	Tigers	2
Giants	Sun	Carp	4
Dragons	Mon	Carp	6
Tigers	Mon	Bay Stars	5
Carp	Mon	Dragons	3
Swallows	Mon	Giants	0
Bay Stars	Mon	Tigers	7
Giants	Mon	Swallows	5

Team/Day pairs such that the team scored the minimum number of runs for that day.

Example

- Using the any or all operations, find the name and age of the oldest student(s)

Example

- Using the any or all operations, find the name and age of the oldest student(s)

```
SELECT sname, age
FROM student s2
WHERE s2.age >= all (SELECT age
                     FROM student s1)
```

You can rewrite queries that use any or all with queries that use exist or not exist

Clicker Question

- Consider the following SQL query

```
SELECT DISTINCT s1.sname, s1.age
FROM student s1, student s2
WHERE s1.age > s2.age
```

- This query returns
- A: The name and age of one of the oldest student(s)
- B: The name and age of all of the oldest student(s)
- C: The name and age of all of the youngest student(s)
- D: The name and age of all students that are older than the youngest student(s)
- E: None of the above

Clicker Question

- Consider the following SQL query

```
SELECT DISTINCT s1.sname, s1.age
FROM student s1, student s2
WHERE s1.age > s2.age
```

- This query returns
- A: The name and age of one of the oldest student(s)
- B: The name and age of all of the oldest student(s)
- C: The name and age of all of the youngest student(s)
- D: The name and age of all students that are older than the youngest student(s)
- E: None of the above

Division in SQL

Find students who've taken all classes.

(method 1)

```
SELECT sname
FROM   Student S
WHERE NOT EXISTS
          ((SELECT  C.name
            FROM  Class C)
            EXCEPT
             (SELECT  E.cname
              FROM  Enrolled E
              WHERE e.snum=S.snum))
```

All classes

Classes taken by S

The hard way (without EXCEPT: (method 2)

```
SELECT sname
FROM    Student S
WHERE NOT EXISTS (SELECT  C.name
                  FROM  Class C
                  WHERE  NOT EXISTS (SELECT  E.snum
                                     FROM  Enrolled E
                                     WHERE  C.name=E.cname
                                     AND E.snum=S.snum))
```

Method 2:

select Student S such that ...
there is no Class C…

which is not taken by S

You're Now Leaving the World of Relational Algebra

- You now have many ways of asking relational algebra queries
  - For this class, you should be able write queries using all of the different concepts that we've discussed & know the terms used
  - In general, use whatever seems easiest, unless the question specifically asks you to use a specific method.
  - Sometimes the query optimizer may do poorly, and you'll need to try a different version, but we'll ignore that for this class.

Mind the gap

- But there's more you might want to know!
- E.g., "find the average age of students"
- There are extensions of Relational Algebra that cover these topics
  - We won't cover them
- We will cover them in SQL

Aggregate Operators

- 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

- The following versions eliminate duplicates before applying the operation to attribute A:

  COUNT (DISTINCT A)
  SUM (DISTINCT A)
  AVG (DISTINCT A)

```
SELECT count(distinct s.snum)
FROM enrolled e, Student S
WHERE e.snum = s.snum
```

```
SELECT count(s.snum)
FROM enrolled e, Student S
WHERE e.snum = s.snum
```

Aggregate Operators: Examples

students

```
SELECT  COUNT(*)
FROM    Student
```

Find name and age of the oldest student(s)

```
SELECT  Sname, age
FROM    Student S
WHERE  S.age= (SELECT  MAX(S2.age)
               FROM  Student S2)
```

Finding average age of SR students

```
SELECT  AVG (age)
FROM   Student
WHERE  standing='SR'
```

Aggregation examples

- Find the minimum student age

```
SELECT min(age)
FROM student;
```

- How many students have taken a class with “Database” in the title

```
SELECT count(distinct snum)
FROM enrolled
WHERE cname like '%Database%'
```

GROUP BY and HAVING

- Divide tuples into groups and apply aggregate operations to each group.
- Example: *Find the age of the youngest student for each major.*

For *i* = 'Computer Science',
'Civil Engineering'…

```
SELECT  MIN (age)
FROM    Student
WHERE   major = i
```

- Problem:

  We don't know how many majors exist, not to mention this is not good practice

Grouping Examples

Find the age of the youngest student who is at least 19, for each major

```
SELECT    major, MIN(age)
FROM      Student
WHERE     age >= 19
GROUP BY  major
```

Snum	Major	Age
115987938	Computer Science	20
112348546	Computer Science	19
280158572	Animal Science	18
351565322	Accounting	19
556784565	Civil Engineering	21
…	…	…

No Animal Science

Major	Age
Computer Science	19
Accounting	19
Civil Engineering	21
…	…

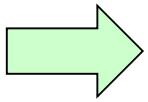
Grouping Examples with Having

Find the age of the youngest student who is at least 19, for each major with at least 2 such students

```
SELECT    major, MIN(age)
FROM      Student
WHERE     age >= 19
GROUP BY  major
HAVING  COUNT(*) > 1
```

Snum	Major	Age
115987938	Computer Science	20
112348546	Computer Science	19
280158572	Animal Science	18
351565322	Accounting	19
556784565	Civil Engineering	21
…	…	…

Major	Age
Computer Science	19
Accounting	19
Civil Engineering	21
…	…



Major	
Computer Science	19

And there are rules

Find the age of the youngest student who is at least 19, for each major with at least 2 such students

```
SELECT    major, MIN(age)
FROM      Student
WHERE     age >= 19
GROUP BY  major
HAVING  COUNT(*) > 1
```

- Would it make sense if I select age instead of MIN(age)?
- *Would it make sense if I select snum to be returned?*
- *Would it make sense if I select major to be returned?*

Major	Age
Computer Science	19
Accounting	19
Civil Engineering	21
…	…

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.

Conceptual Evaluation of a Query

1. compute the cross-product of *relation-list*
2. keep only tuples that satisfy *qualification* where
3. partition the remaining tuples into groups by the value of attributes in *grouping-list*
4. keep only the groups that satisfy *group-qualification* (expressions in *group-qualification* must have a *single value per group*!)
5. delete fields that are not in *target-list*
6. generate one answer tuple per qualifying group.

GROUP BY and HAVING (cont)

- Example1: *For each class, find the age of the youngest student who has enrolled in this class:*

```
SELECT      cname,  MIN(age)
FROM        Student S, Enrolled E
WHERE       S.snum= E.snum
GROUP BY    cname
```

- Example2: *For each course with more than 1 enrollment, find the age of the youngest student who has taken this class:*

```
SELECT      cname,  MIN(age)
FROM        Student S, Enrolled E
WHERE       S.snum = E.snum
GROUP BY    cname
HAVING      COUNT(*) > 1
```

← per group qualification!