INFS7901
Database Principles

Structured Query Language (SQL)

Rocky Chen

Aggregation, GROUP BY and HAVING

INSERT, DELETE and UPDATE statements

Nested Queries

Views

Aggregation in SQL

- Aggregates are functions that produce summary values.

```
SELECT [DISTINCT] (attribute / exprsn / aggregation-function list | * )
FROM <table list>
[WHERE [join condition and]    search_condition]
[ORDER BY column_name [ASC|DESC] {, column-name [ASC|DESC]}];
```

- The aggregation-function list may include:
  - **SUM/AVG** ([DISTINCT] expression): Calculates the sum/ average of a set of *numeric* values
  - **COUNT** ([DISTINCT] expression): Counts the number of tuples that the query returns
  - **COUNT**(*)
  - **MAX/MIN**(expression): Returns the maximum (minimum) value from a set of values which have a *total ordering*. Note that the domain of values can be non-numeric.

Aggregate Operators Examples

College(cName, state, enrollment)
Student(sID, sName, GPA, sizeHS)
Apply(sID, cName, major, decision)

# students

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

Finding average GPA of students from high schools with less than 500 students

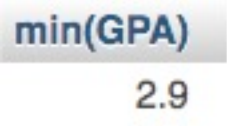
```
SELECT  AVG (GPA)
FROM    Student
WHERE   sizeHS<500
```

Aggregation Examples

College(cName, state, enrollment)
Student(sID, sName, GPA, sizeHS)
Apply(sID, cName, major, decision)

- Find the minimum GPA.

```
SELECT min(GPA)
FROM Student
```



```
min(GPA)
     2.9
```

- Find how many students have applied to 'Stanford'.

```
SELECT count(distinct sID)
FROM Apply
where cname like 'Stanford'
```

Note: want DISTINCT when Students apply to more than one major at Stanford

GROUP BY and HAVING

- Divide tuples into groups and apply aggregate operations to each group.

- Example: Find the enrollment of the smallest college from each state

College(cName, state, enrollment)
Student(sID, sName, GPA, sizeHS)
Apply(sID, cName, major, decision)

GROUP BY Syntax

- Aggregation functions can also be applied to groups of rows within a table. The GROUP BY clauses provides this functionality.

```
SELECT [DISTINCT] (attribute / expression / aggregation-function list | * )
FROM <table list>
[WHERE [join condition and]      search_condition]
[GROUP BY grouping attributes]
[ORDER BY column_name [ASC|DESC] {, column-name [ASC|DESC]}];
```

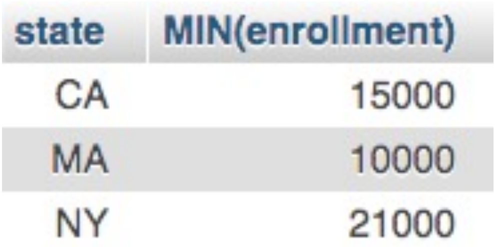
- When GROUP BY is used in an SQL statement, any attribute appeared in SELECT clause must also appeared in an aggregation function or in GROUP BY clause.

Grouping Examples

- Example: Find the enrollment of the smallest college from each state

College(cName, state, enrollment)
Student(sID, sName, GPA, sizeHS)
Apply(sID, cName, major, decision)

```
SELECT state,  MIN(enrollment)
FROM College
GROUP BY state
```



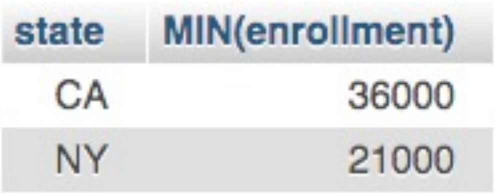
state	MIN(enrollment)
CA	15000
MA	10000
NY	21000

Grouping Examples

- Example: *Find the enrollment of the smallest college from each state which has more than 15000 students.*

College(cName, state, enrollment)
Student(sID, sName, GPA, sizeHS)
Apply(sID, cName, major, decision)

```
SELECT state,  MIN(enrollment)
FROM College
WHERE enrollment > 15000
GROUP BY state
```



state	MIN(enrollment)
CA	36000
NY	21000

Conditions on Groups

- Conditions can be imposed on the selection of groups to be included in the query result.
- The HAVING clause (following the GROUP BY clause) is used to specify these conditions, similar to the WHERE clause.

```
SELECT [DISTINCT] (attribute / expression / aggregation-function list | * )
FROM <table list>
[WHERE [join condition and] search_condition]
[GROUP BY grouping attributes]
[HAVING <group condition>]
[ORDER BY column_name [ASC|DESC] {, column-name [ASC|DESC]}];
```

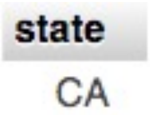
- Unlike the WHERE clause, the HAVING clause can also include aggregates.

Grouping Examples with Having

- *Find states that have more than one college.*

College(cName, state, enrollment)
Student(sID, sName, GPA, sizeHS)
Apply(sID, cName, major, decision)

```
SELECT      state
FROM        College
GROUP BY   state
HAVING  COUNT(*) > 1
```



state
CA

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 of an aggregate operator.

Conceptual Evaluation of a Query

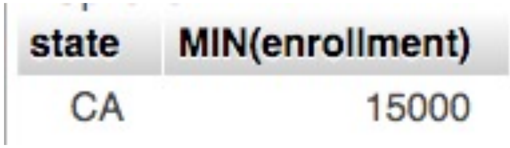
1. compute the cross-product of *relation-list.*
2. keep only tuples that satisfy *qualification.*
3. partition the remaining tuples into groups by where attributes in *grouping-list*.
4. keep only the groups that satisfy *group-qualification* (expressions in *group-qualification* must have a *single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

Grouping Examples

- Find the enrollment of the smallest college with enrollment >10000 for each state with at least 2 colleges (of enrollment >10000)

College(cName, state, enrollment)
Student(sID, sName, GPA, sizeHS)
Apply(sID, cName, major, decision)

```
SELECT  state,  MIN(enrollment)
FROM     College
WHERE   enrollment > 10000
GROUP BY  state
HAVING  count(*)  > 1
```



state	MIN(enrollment)
CA	15000

Clicker Question on Grouping

- Compute the result of the query:

```
SELECT a1.x, a2.y, COUNT(*)
FROM     Arc a1, Arc a2
WHERE a1.y = a2.x
GROUP BY a1.x, a2.y
```

Which of the following is in the result?

A. (1,3,2)

B. (4,2,6)

C. (4,3,1)

D. All of the above

E. None of the above

x	y
1	2
1	2
2	3
3	4
3	4
4	1
4	1
4	1
4	2

clickergrouping.sql

Clicker Question on Grouping

- Compute the result of the query:

```
SELECT a1.x, a2.y, COUNT(*)
FROM    Arc a1, Arc a2
WHERE a1.y = a2.x
GROUP BY a1.x, a2.y
```

Which is in the result?

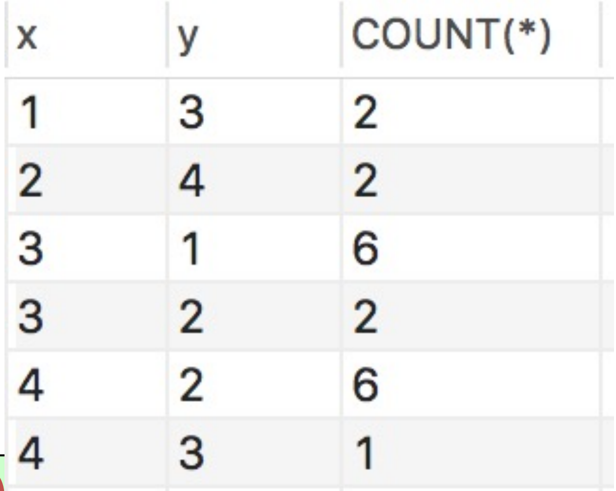
A. (1,3,2) (1,2)(2,3), (1,2)(2,3)

B. (4,2,6) 3 ways to do (4,1) and two ways to do (1,2)

C. (4,3,1) (4,2)(2,3)

D. All of the above Correct

E. None of the above



x	y	COUNT(*)
1	3	2
2	4	2
3	1	6
3	2	2
4	2	6
4	3	1

x	y
1	2
1	2
2	3
3	4
3	4
4	1
4	1
4	1
4	2

Tip: You can think of Arc as being a flight, and the query as asking for how many ways you can take each 2 hop plane trip

Aggregation, GROUP BY and HAVING

INSERT, DELETE and UPDATE statements

Nested Queries

Views

INSERT Statement

- INSERT statement is used to add tuples to an existing relation
- Single Tuple INSERT
  - Specify the relation name and a list of values for the tuple
  - Values are listed in the same order as the attributes were specified in the CREATE TABLE command
  - User may specify explicit attribute names that correspond to the values provided in the insert statement. The attributes not included cannot have the NOT NULL constraint
- Multiple Tuple INSERT
  - By separating each tuple’s list of values with commas
  - By loading the result of a query

Single Tuple INSERT Example

```
INSERT INTO <table name>
        [(<column name> {, <column name> })]
(VALUES (<constant value>, {,<constant value> })
                | <select statement>);
```

Student(sID, sName, GPA, sizeHS)

- Can insert a single tuple using:

```
INSERT  INTO  Student
VALUES  (53688, ‘Smith’, 3.2, 200)
```

- or

```
INSERT  INTO  Student (sID, sName, GPA, sizeHS)
VALUES  (53688, ‘Smith’, 3.2, 200)
```

- Add a tuple to student with null address and phone:

```
INSERT  INTO  Student (sID, sName, GPA, sizeHS)
VALUES  (53688, ‘Smith’, 3.2, NULL)
```

Multiple Tuple INSERT Example

```
INSERT INTO <table name>
      [(<column name> {, <column name> })]
(VALUES (<constant value>, {,<constant value> })
                  | <select statement>);
```

Apply(sID, cName, major, decision)

- Can add values selected from another table
- Make student 123 apply into all “BIO” related majors at Stanford.

```
INSERT  INTO  apply
      SELECT  123, "Stanford",  major, NULL
       FROM apply
         WHERE major LIKE "%bio%"
```

The select-from-where statement is fully evaluated before any of its results are inserted or deleted.

DELETE Statement

- DELETE statement is used to remove existing tuples from a relation.
- A single DELETE statement may delete zero, one, several or all tuples from a table.
- Tuples are explicitly deleted from a single table.
- Deletion may propagate to other tables if referential triggered actions are specified in the referential integrity constraints of the CREATE (ALTER) TABLE statement

```
DELETE FROM <table name>
     [WHERE <select condition>];
```

DELETE Statement Example

- Delete all “BIO” related applications of student 123 for Stanford.

Apply(sID, cName, major, decision)

```
DELETE FROM  apply
WHERE sID = 123 AND
cName LIKE "Stanford" AND major LIKE "%BIO%"
```

- Note that only whole tuples are deleted.

- Can delete all tuples satisfying some condition

UPDATE Statement

UPDATE statement is used to modify attribute values of one or more selected tuples in a relation.

- Tuples are selected for update from a single table.
- However, updating a primary key value may propagate to other tables if referential triggered actions are specified in the referential integrity constraints of the CREATE (ALTER) TABLE statement.

```
UPDATE <table name>
    SET <column name> = <value expression>
        {, <column name>  = <value expression>}
    [WHERE <select condition>];
```

UPDATE Statement Example

- Increase the high school size of all students by 5 (should not be more than 500)

Student(sID, sName, GPA, sizeHS)

- Need to write two updates:

```
UPDATE  Student
SET         sizeHS = 500
WHERE   sizeHS >= 495

UPDATE  Student
SET sizeHS = sizeHS + 5
WHERE sizeHS < 495
```

- Is the order important?

Aggregation, GROUP BY and HAVING

INSERT, DELETE and UPDATE statements

Nested Queries

Views

Motivating Example for Nested Queries

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

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

- Find ids and names of stars who have not been in movie with ID 28:

```
~~SELECT  Distinct M.StarID, name~~
~~FROM    MovieStar M, StarsIn S~~
~~WHERE   M.StarID = S.starID AND S.MovieID <> 28;~~
```

Nested SQL Queries: What are They?

- A nested query (often termed sub-query) is a query that appears within another query.
  - Inside the WHERE clause of another SELECT statement.
  - Inside an INSERT, UPDATE or DELETE statement.
  - Nesting can occur at multiple levels.
- Nested queries are useful for expressing queries where data must be fetched and used in a comparison condition.

Non-correlated Nested Queries

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

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

TRY "NOT IN"

- In non-correlated nested queries, the inner query does not depend on the outer query, and the outer query directly takes an action based on the results of the inner query.

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

StarID
1026
1027

```
SELECT  M.StarID, M.Name
FROM    MovieStar M
WHERE   M.StarID IN
(1026,1027)
```

Correlated Nested Queries

- Correlated Nested Queries
  - Correlated subqueries have conditions in their WHERE clause that references some attribute of a relation declared in the outer query.
  - The outer SQL statement provides the values for the inner subquery to use in its evaluation.
  - The subquery is evaluated once for each (combination of) tuple in the outer query.

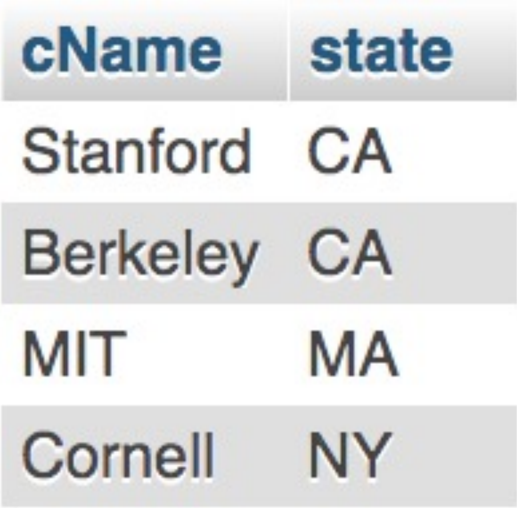
Correlated Nested Queries Example

- For each college, check if there is another college in the same state

```
SELECT  cName,  state
FROM  College C1
WHERE  exists  (SELECT  *
                    FROM  College C2
                     WHERE C2.state = C1.state AND
              C2.cName <> C1.cName);
```

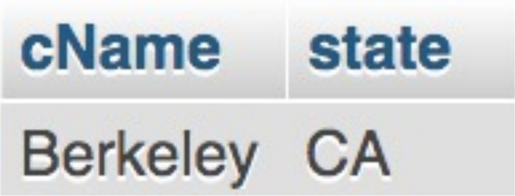
Think of this as passing parameters

Outer Query



cName	state
Stanford	CA
Berkeley	CA
MIT	MA
Cornell	NY

Results



cName	state
Berkeley	CA

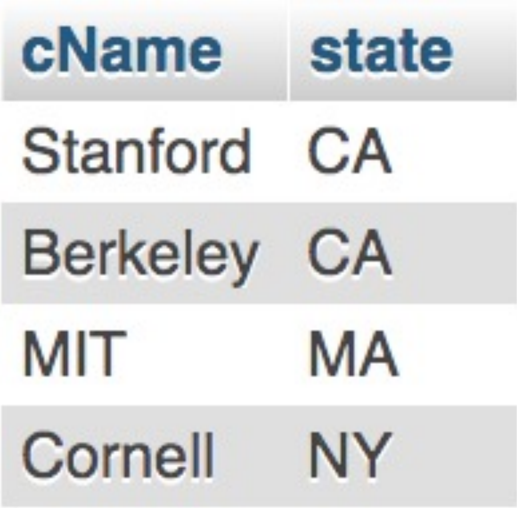
Correlated Nested Queries Example

- For each college, check if there is another college in the same state

```
SELECT cName, state
FROM College C1
WHERE exists (SELECT *
                FROM College C2
                 WHERE C2.state = C1.state AND
          C2.cName <> C1.cName);
```

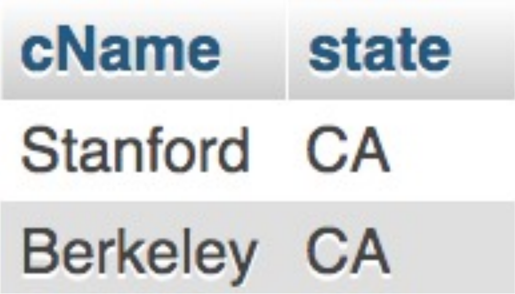
Think of this as passing parameters

Outer Query



cName	state
Stanford	CA
Berkeley	CA
MIT	MA
Cornell	NY

Results



cName	state
Stanford	CA
Berkeley	CA

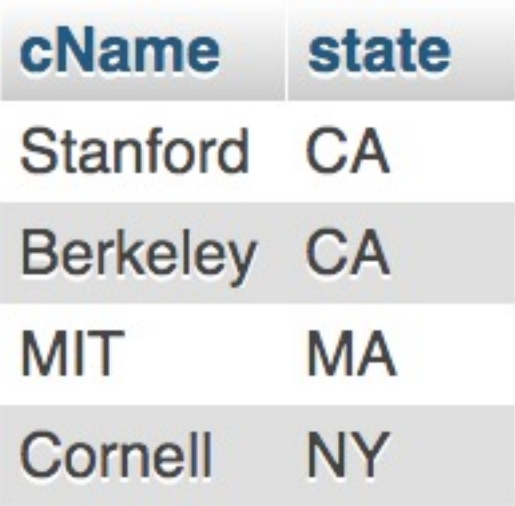
Correlated Nested Queries Example

- For each college, check if there is another college in the same state

```
SELECT  cName,  state
FROM  College C1
WHERE  exists  (SELECT  *
                    FROM  College C2
                     WHERE C2.state = C1.state AND
              C2.cName <> C1.cName);
```

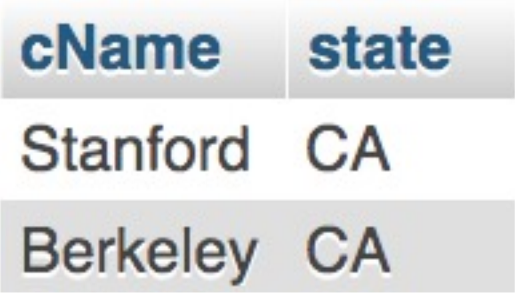
Think of this as passing parameters

Outer Query



cName	state
Stanford	CA
Berkeley	CA
MIT	MA
Cornell	NY

Results



cName	state
Stanford	CA
Berkeley	CA

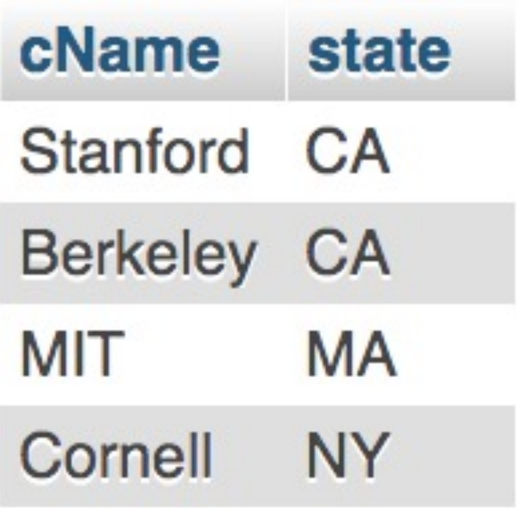
Correlated Nested Queries Example

- For each college, check if there is another college in the same state

```
SELECT cName, state
FROM College C1
WHERE exists (SELECT *
                 FROM College C2
                  WHERE C2.state = C1.state AND
           C2.cName <> C1.cName);
```

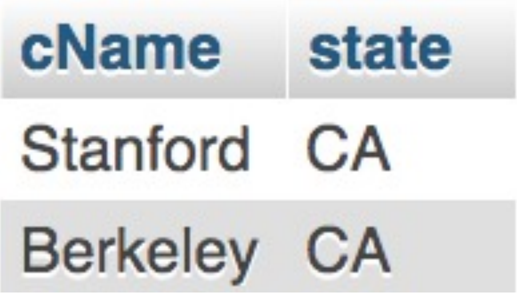
Think of this as passing parameters

Outer Query



cName	state
Stanford	CA
Berkeley	CA
MIT	MA
Cornell	NY

Results



cName	state
Stanford	CA
Berkeley	CA

Sub-query Operators

- Sub-queries that return a set
  - expression {[NOT] IN (*sub-query*)
  - expression comp-op [ANY|ALL] (*sub-query*)

- Subqueries that return a single value
  - expression comp-op (sub-query)

Sub-query Returning a Set

Expression and attribute list in sub-query SELECT clause must have same domain

- expression {[NOT] IN (*sub-query*)
  - expression is checked for **membership** in the set (of tuples) returned by sub-query.
- expression comp-op [ANY|ALL] (*sub-query*)
  - expression is **compared** with the set (of tuples) returned by the sub-query
  - ANY: Evaluates to true if one comparison is true
  - ALL: Evaluates to true if all comparisons are true

IN/NOT IN Operator Example

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

```
SELECT  M.StarID, M.Name
FROM    MovieStar M
WHERE   M.StarID NOT IN  (SELECT  S.StarID
                     FROM   StarsIn S
                     WHERE  MovieID=28)
```

ANY/ ALL Operator Example

- 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

- Assuming we have multiple movies named Fargo, how would the use of ALL vs. ANY affect the result?

Equivalence of IN and = ANY

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

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

```
SELECT  M.StarID, M.Name
FROM    MovieStar M
WHERE  M.StarID = ANY  (SELECT  S.StarID
                      FROM  StarsIn S
                      WHERE  MovieID=28)
```

Equivalence of Not IN and <>ALL

- Find ids and names of stars who have NOT been in movie with ID 28:

```
SELECT  M.StarID, M.Name
FROM    MovieStar M
WHERE   M.StarID NOT IN  (SELECT  S.StarID
                          FROM   StarsIn S
                          WHERE  MovieID=28)
```

```
SELECT  M.StarID, M.Name
FROM    MovieStar M
WHERE   M.StarID <> ALL  (SELECT  S.StarID
                          FROM   StarsIn S
                          WHERE  MovieID=28)
```

Non-Equivalence of Not IN and <>ANY

- If a sub-query returns: {$30K, $32K, $37K}

- NOT IN means
  - – NOT=$30K AND NOT=$32K AND NOT=$37K

- <> ANY means
  - – NOT=$30K *OR* NOT=$32K *OR* NOT=$37K

<> ANY will be true for any value in this example.

Clicker Nested Question

Consider the following table and SQL query:

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

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

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

Clicker Nested Question

Clickernested.sql

Consider the following table and SQL query:

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

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

Which of the following is in the result:

A. (Carp, Sun)

B. (Bay Stars, Sun)

C. (Swallows, Mon)

D. All of the above Correct

E. None of the above

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

Sub-query Returning a Set

- The select list of an inner sub-query introduced with a comparison operator (and ANY/ALL) or IN can include only one expression or column name.

- The expression you name in the WHERE clause of the outer statement must be join compatible with the column you name in the sub-query select list.

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

Sub-query Returning a Value

Expression and attribute list in sub-query SELECT clause must have same domain.

- **expression comp-op** (sub-query)
  - expression is **compared** with the *value* returned by the sub-query.
  - The sub-query must evaluate to a single value otherwise an error will occur.

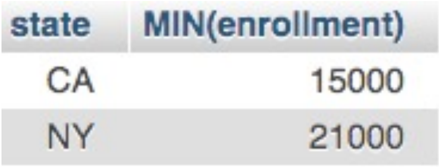
```
SELECT Title
FROM Movie
Where year = (SELECT max(year)
              FROM Movie)
```

Nested Grouping Example 1

College(cName, state, enrollment)
Student(sID, sName, GPA, sizeHS)
Apply(sID, cName, major, decision)

- Find the enrollment of the smallest college with enrollment >10000 for each state with average enrollment higher than the average enrollment across all of the colleges

```
SELECT  state,  MIN(enrollment)
FROM     College
WHERE   enrollment > 10000
GROUP BY  state
HAVING  avg(enrollment)  > (SELECT avg(enrollment)
                             FROM College)
```

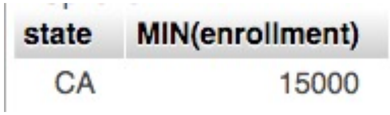


state	MIN(enrollment)
CA	15000
NY	21000

Nested Grouping Example 2

- Find the enrollment of the smallest college with enrollment >10000 for each state with at least 2 colleges.

```
SELECT  state,  MIN(enrollment)
FROM     College C1
WHERE   enrollment > 10000
GROUP BY  state
HAVING  1 < (SELECT count(*)
               FROM College C2
               WHERE C1.state = C2.state)
```



state	MIN(enrollment)
CA	15000

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

Using the Exists Function

- The EXISTS function tests for the existence or nonexistence of data that meet the criteria of the sub-query

```
SELECT … FROM …
WHERE [NOT] EXISTS (sub-query)
```

- Sub-queries are used with EXISTS and NOT EXISTS are always correlated
  - WHERE EXISTS (sub-query) evaluates to true if the result of the correlated sub-query is a non-empty set, i.e. contains 1 or more tuples.
  - WHERE NOT EXISTS (sub-query) evaluates to true if the result of the correlated sub-query returns an empty set, i.e. zero tuples.

NOT EXISTS Example

- Find movies that were the only movie of the year.

```
SELECT  *
From Movie M1
where NOT EXISTS
     (Select *
      FROM Movie M2
      Where M1.movieID <> M2.movieID and M1.year = M2.year)
```

- Find movies that were not the only movie of the year.

```
SELECT  *
From Movie M1
where EXISTS
     (Select *
      FROM Movie M2
      Where M1.movieID <> M2.movieID and M1.year = M2.year)
```

Sub-query using ORDER BY

- Sub-queries cannot include the ORDER BY clause. The optional DISTINCT keyword may effectively order the results of a sub-query, since most systems eliminate duplicates by first ordering the results

Sub-queries vs. Set Operations and Joins

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

```
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
```

```
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)
```

Sub-queries vs. Set Operations and Joins

- Find IDs of stars who have been in movies in 1944 or 1974.

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

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

Sub-queries vs. Set Operations in Difference

- Find IDs of stars who have been in movies in 1944 and not in1974.

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

```
SELECT S.StarID
FROM    Movie M, StarsIn S
WHERE  M.MovieID = S.MovieID AND M.year = 1944 AND
 S.StarID NOT IN (SELECT  S2.StarID
              FROM Movie M2, StarsIn S2
              WHERE   M2.MovieID = S2.MovieID AND M2.year = 1974)
```

Nested Queries Example

- Find IDs and names of students applying to CS (using both join and nested queries).

College(cName, state, enrollment)
Student(sID, sName, GPA, sizeHS)
Apply(sID, cName, major, decision)

```
SELECT sID, sName
FROM Student
WHERE sID in (SELECT sID
                  FROM Apply
              WHERE major = 'CS');
```

```
SELECT DISTINCT Student.sID, sName
FROM Student, Apply
WHERE Student.sID = Apply.sID
 and major = 'CS';
```

Nested Query Example (Tricky)

- Find names of students applying to CS (using both join and nested queries).

```
College(cName, state, enrollment)
Student(sID, sName, GPA, sizeHS)
Apply(sID, cName, major, decision)
```

```
SELECT sName
FROM Student
WHERE sID in (SELECT sID
              FROM Apply
              WHERE major = 'CS');
```

```
SELECT sName
FROM Student, Apply
WHERE Student.sID = Apply.sID
and major = 'CS';
```

Both with and without distinct is incorrect – think about Names

Why are Duplicates Important?

- Find GPA of CS applicants (using both join and nested queries)

College(cName, state, enrollment)
Student(sID, sName, GPA, sizeHS)
Apply(sID, cName, major, decision)

```
SELECT GPA
FROM Student
WHERE sID in (SELECT sID
              FROM Apply
              WHERE major = 'CS');
```

```
SELECT GPA
FROM Student, Apply
WHERE Student.sID = Apply.sID
 and major = 'CS';
```

Both with and without distinct is incorrect

Joins vs Sub-queries

- Many nested queries are equivalent to a simple query using JOIN operation. However, in many cases, the use of nested queries is necessary and cannot be replaced by a JOIN operation.

- Recommendation
  - Use joins when you are displaying results from multiple tables.
  - Use sub-queries when you need to compare aggregates to other values.

Division in SQL

- Division in SQL is useful for answering queries include a “**for all**” or “**for every**” phrase, e.g., Find movie stars who were in **all** movies.

- Unfortunately, there is no direct way to express division in SQL. We can write this query, but to do so, we will have to express our query through double negation and existential quantifiers.

Examples of Division A/B

A

sno	pno
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

B1

pno
p2

B2

pno
p2
p4

B3

pno
p1
p2
p4

A/B1

sno
s1
s2
s3
s4

A/B2

sno
s1
s4

A/B3

sno
s1

Division in SQL Using EXCEPT

- Find the IDs of movie stars who have played in all of the movies.

Movie(MovieID, Title, Year)
StarsIn(MovieID, StarID, Role)
MovieStar(StarID, Name, Gender)

```
             SELECT StarID
             FROM   MovieStar MS
             WHERE NOT EXISTS
All movies              ((SELECT  MovieID
                          FROM  Movies)
Movies                  EXCEPT
Played by MS            (SELECT  MovieID
                        FROM  StarsIn S
                        WHERE  S.StarID=MS.StarID))
```

Division in SQL Using NOT EXISTS

- Find the IDs of movie stars who have played in all of the movies.

Movie(MovieID, Title, Year)
StarsIn(MovieID, StarID, Role)
MovieStar(StarID, Name, Gender)

```
SELECT StarID
FROM    MovieStar MS
WHERE NOT EXISTS
        (SELECT  M.MovieID
        FROM  Movie M
        WHERE  NOT EXISTS
            (SELECT  S.MovieID
            FROM  StarsIn S
            WHERE  S.MovieID=M.MovieID
            AND S.StarID=MS.StarID))
```

select Movie Star MS such that
  there is no Movie M…
    which is not played by MS

Aggregation, GROUP BY and HAVING

INSERT, DELETE and UPDATE statements

Nested Queries

Views

Motivating Example for Use of Views

- Find those states for which their average college enrollment is the minimum over all states.

College(cName, state, enrollment)
Student(sID, sName, GPA, sizeHS)
Apply(sID, cName, major, decision)

~~SELECT state, avg(enrollment)~~
~~FROM College~~
~~GROUP BY state~~
~~HAVING min(avg(enrollment))~~

Wrong, cannot use nested aggregation

- One solution would be to use subquery in the FROM Clause.

```
SELECT  Temp.state, Temp.average
FROM(SELECT state, AVG(enrollment) as average
        FROM  College
        GROUP BY state) AS Temp
WHERE Temp.average in (SELECT  MIN(Temp.average) FROM  Temp)
```

Hideously ugly
Not supported
in all systems

- A Better alternative is to use views

What Are Views?

- A View is a single table that is derived from other tables, which could be base tables or previously defined views
- Views can be
  - Virtual tables - that do not physically exist on disk.
  - Materialized - by physically creating the view table. These must be updated when the base tables are updated
- We can think of a virtual view as a way of specifying a table that we need to reference frequently, even though it does not physically exist.

Benefits of Using Views

- **Simplification**: View can hide the complexity of underlying tables to the end-users.
- **Security**: Views can hide columns containing sensitive data from certain groups of users.
- **Computed columns**: Views can create computed columns, which are computed on the fly.
- **Logical Data Independence**: Views provide support for logical data independence, that is users and user's programs that access the database are immune from changes in the logical structure of the database.

Defining and Using Views

```
CREATE VIEW <view name>
      (<column name> {, <column name>}) AS
       <select statement> ;
```

- Example: Suppose we have the following 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).

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

- 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.

View Updates

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

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

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

Example: UQ 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.

Dropping Views

```
DROP VIEW [IF EXISTS]
view_name [, view_name] ...
 [RESTRICT | CASCADE]
```

- Dropping a view does not affect any tuples of the underlying relation.

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

The Beauty of Views

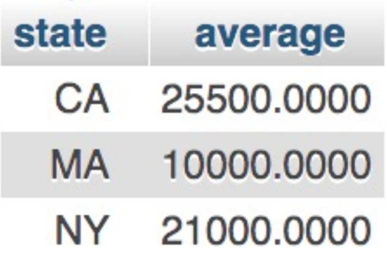
- Find those states for which their average college enrollment is the minimum over all states.

```
SELECT  Temp.state, Temp.average
FROM(SELECT state, AVG(enrollment) as average
        FROM  College
        GROUP BY state) AS Temp
WHERE Temp.average in (SELECT  MIN(Temp.average) FROM  Temp)
```

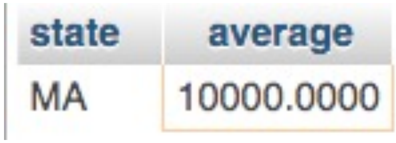
Hideously ugly
Not supported
in all systems

```
Create View Temp(state, average) as
            SELECT state, AVG(enrollment) AS average
            FROM College
            GROUP BY  state;

Select state, average
From Temp
WHERE average = (SELECT MIN(average) from Temp)
```



state	average
CA	25500.0000
MA	10000.0000
NY	21000.0000



state	average
MA	10000.0000

Clicker Question on Views

Consider the following table and SQL queries:

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

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

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 on Views

Consider the following table and SQL queries:

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

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

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) Wrong. "Count"

B. (3,12)

C. (5,9) Correct

D. All are correct

E. None are correct

Clickerview.sql