CS 4604: Introduction to Database Management Systems

B. Aditya Prakash

Lecture #3: SQL and Relational Algebra---Part 1
Reminder: Relational Algebra

- Relational algebra is a notation for specifying queries about the contents of relations

- Notation of relational algebra eases the task of reasoning about queries

- Operations in relational algebra have counterparts in SQL
What is an Algebra?

- An algebra is a set of operators and operands
  - Arithmetic: operands are variables and constants, operators are +, -, *, ÷, /, etc.
  - Set algebra: operands are sets and operators are ∩, U, -

- An algebra allows us to
  - construct expressions by combining operands and expression using operators
  - has rules for reasoning about expressions

\[ a^2 + 2 \times a \times b + 2b, \quad (a + b)^2 \]
\[ R - (R - S), \quad R \cap S \]
Relational operators

- selection
- Projection
- set union
- set difference

\[ \sigma_{condition} (R) \]

\[ \pi_{att-list} (R) \]

R U S

R - S
Clarification: Projection

- The projection operator produces from a relation R a new relation containing only **some of R’s columns**

- “Delete” (i.e. not show) attributes not in projection list

- Duplicates eliminated (sets vs **multisets**)  

- To obtain a relation containing only the columns $A_1,A_2, \ldots, A_n$ of R

  $RA: \quad \pi \ A_1,A_2, \ldots, A_n (R)$

  $SQL: \quad SELECT \ A_1,A_2, \ldots, A_n \ FROM \ R;$
Clarification: Projection

- The projection operator produces from a relation R a new relation containing only some of R’s columns.
- “Delete” (i.e. not show) attributes not in projection list.
- Duplicates eliminated (sets vs multisets).

To obtain a relation containing only the columns A₁,A₂, . . . Aₙ of R:

RA: \[ \pi \ A₁,A₂, \ldots \ Aₙ \ (R) \]

SQL: \[ \text{SELECT DISTINCT A₁,A₂, \ldots Aₙ FROM R; } \]
What about strings?

find student ssns who live on "main" (st or str or street - ie., "main st" or "main str" ...)
What about strings?

find student ssnss who live on “main” (st or str or street)

    select ssn
    from student
    where address like “main%”

%: variable-length don’t care
_: single-character don’t care
Relational operators

Are we done yet?

Q: Give a query we can not answer yet!
Relational operators

A: any query across two or more tables, eg., ‘find names of students in 4604’

Q: what extra operator do we need??

<table>
<thead>
<tr>
<th>STUDENT</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ssn</td>
<td>Name</td>
<td>Address</td>
</tr>
<tr>
<td>123</td>
<td>smith</td>
<td>main str</td>
</tr>
<tr>
<td>234</td>
<td>jones</td>
<td>forbes ave</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>4604</td>
<td>A</td>
</tr>
<tr>
<td>234</td>
<td>3114</td>
<td>B</td>
</tr>
</tbody>
</table>
Relational operators

A: any query across two or more tables, eg., ‘find names of students in 4604’

Q: what extra operator do we need??

A: surprisingly, cartesian product is enough!

<table>
<thead>
<tr>
<th>STUDENT</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ssn</td>
<td>Name</td>
<td>Address</td>
</tr>
<tr>
<td>123</td>
<td>smith</td>
<td>main str</td>
</tr>
<tr>
<td>234</td>
<td>jones</td>
<td>forbes ave</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>4604</td>
<td>A</td>
</tr>
<tr>
<td>234</td>
<td>3114</td>
<td>B</td>
</tr>
</tbody>
</table>
The Cartesian product (or cross-product or product) of two relations R and S is a the set of pairs that can be formed by pairing each tuple of R with each tuple of S.

- The result is a relation whose schema is the schema for R followed by the schema for S.

**RA:** \( R \times S \)

**SQL:** `SELECT * FROM R, S ;`
## Cartesian Product

**S1**

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

**R1**

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

**S1 X R1**

<table>
<thead>
<tr>
<th>(sid)</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>(sid)</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>31</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

We rename attributes to avoid ambiguity or we prefix attribute with the name of the relation it belongs to.
FUNDAMENTAL
Relational operators

- selection: $\sigma_{\text{condition}}(R)$
- projection: $\pi_{\text{att-list}}(R)$
- cartesian product: $R \times S$
- set union: $R \cup S$
- set difference: $R - S$
Relational ops

- Surprisingly, they are enough, to help us answer almost any query we want!
- derived/convenience operators:
  - set intersection  --- (We have seen this)
  - **join** (theta join, equi-join, natural join)
  - ‘rename’ operator \( \rho_{R'}(R) \)
  - division \( R \div S \)
Theta-Join

- The theta-join of two relations R and S is the set of tuples in the Cartesian product of R and S that satisfy some condition C.

RA: \( R \bowtie_C S \)

SQL: 

```sql
SELECT * 
FROM R, S 
WHERE C;
```

- \( R \bowtie_C S = \sigma_C (R \times S) \)
Theta-Join

\[
S1 \bowtie_{S1.\text{sid}<R1.\text{sid}} R1
\]

\[
\begin{array}{|c|c|c|c|}
\hline
\text{sid} & \text{sname} & \text{rating} & \text{age} \\
\hline
22 & dustin & 7 & 45.0 \\
31 & lubber & 8 & 55.5 \\
58 & rusty & 10 & 35.0 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
\text{sid} & \text{bid} & \text{day} \\
\hline
22 & 101 & 10/10/96 \\
58 & 103 & 11/12/96 \\
\hline
\end{array}
\]

\[
R \bowtie_{C} S = \sigma_{C} (R \times S)
\]
**Natural Join**

- The natural join of two relations R and S is a set of *pairs of tuples*, one from R and one from S, that agree on whatever attributes are common to the schemas of R and S.
- The schema for the result contains the *union of the attributes of R and S*. (so duplicate cols. are dropped)
- Assume the schemas R(A,B, C) and S(B, C,D)

RA:  \( R \Join S \)

SQL:  \( \text{SELECT } R.A, R.B, R.C, S.D \)

\( \text{FROM } R, S \)

\( \text{WHERE } R.B = S.B \land R.C = S.C; \)
Natural Join: Nit-picking

- What if R and S have not attributes in common?

  natural join $\rightarrow$ cartesian product

- Some (like Oracle) provide a special single NATURAL JOIN operator, but some (like IBM DB2) don’t.
  - So assume there is no special SQL natural join operator
Operators so far

- **Remove parts of single relations**
  - Projection: $\pi_{(A,B)}(R)$ and SELECT A, B FROM R
  - Selection: $\sigma_C(R)$ and SELECT * FROM R WHERE C
  - Combining Projection and Selection:
    - $\pi_{(A,B)}(\sigma_C(R))$
    - SELECT A, B FROM R WHERE C
Operations so far

- **Set operations**
  - R and S must have the same attributes, same attribute types, and same order of attributes
  - Union: R U S and (R) UNION (S)
  - Intersection: R ∩ S and (R) INTERSECT (S)
  - Difference: R – S and (R) EXCEPT (S)
Operations so far

- Combine the tuples of two relations
  - Cartesian Product: \( R \times S, \ldots \) FROM \( R, S \) ..... 
  - Theta Join: \( R \bowtie_C S, \ldots \) FROM \( R, S \) WHERE \( C \) 
  - Natural Join: \( R \bowtie S \)
Ordering

- find student records, sorted in name order
  - select *
  - from student
  - order by name asc

- asc is the default
Ordering

- find student records, sorted in name order; break ties by reverse ssn
  - select *
  - from student
  - order by name, ssn desc
Rename op.

- Q: why? \[ \rho_{AFTER}(BETORE) \]
- A: shorthand; self-joins; ...
- for example, find the grand-parents of ‘Tom’, given PC (parent-id, child-id)
Rename op.

- PC (parent-id, child-id)

```
PC
p-id  | c-id
Mary  | Tom
Peter | Mary
John  | Tom
```

```
PC
p-id  | c-id
Mary  | Tom
Peter | Mary
John  | Tom
```
Rename op.

- first, WRONG attempt:
  \[ PC \bowtie PC \]
  (why? how many columns?)
- Second WRONG attempt:
  \[ PC \bowtie_{PC.c-id=PC.p-id} PC \]
Rename op.

- we clearly need two different names for the same table - hence, the ‘rename’ op.

\[ \rho_{PC1}(PC) \Join_{PC1.c-id=PC.p-id} PC \]
Disambiguation and Renaming

**RA:** give R the name S; R has n attributes, which are $\rho_{S}^{\rho_{S}}(A_1, A_2, \ldots, A_n)$ (R) called A1, A2, . . . , An in S.

**SQL:** Use the **AS** keyword in the **FROM** clause: Students AS Students1 renames Students to Students1.

**SQL:** Use the **AS** keyword in the **SELECT** clause to rename attributes.
Disambiguation and Renaming

- Name pairs of students who live at the same address: **Students (Name, Address)**

**RA:** \[ \pi_{S1.Name, S2.Name} \left( \sigma_{S1.Address = S2.Address} \left( \rho_{S1}(Students) \times \rho_{S2}(Students) \right) \right) \]

**SQL:**
```
SELECT S1.name, S2.name
FROM Students AS S1, Students AS S2
WHERE S1.address = S2.address
```
Name pairs of students who live at the same address:

**SQL:**

```sql
SELECT S1.name, S2.name
FROM Students AS S1, Students AS S2
WHERE S1.address = S2.address
```

**Are these correct?**

**No !!!** the result includes tuples where a student is paired with himself/herself

**Solution:** Add the condition `S1.name <> S2.name`.
Division

- Rarely used, but powerful.
- Example: find suspicious suppliers, i.e., suppliers that supplied all the parts in A_BOMB.
## Division

<table>
<thead>
<tr>
<th>SHIPMENT</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s#</td>
<td></td>
<td>p#</td>
</tr>
<tr>
<td>s1</td>
<td></td>
<td>p1</td>
</tr>
<tr>
<td>s2</td>
<td></td>
<td>p1</td>
</tr>
<tr>
<td>s1</td>
<td></td>
<td>p2</td>
</tr>
<tr>
<td>s3</td>
<td></td>
<td>p1</td>
</tr>
<tr>
<td>s5</td>
<td></td>
<td>p3</td>
</tr>
</tbody>
</table>

\[ \text{ABOMB} \div \begin{array}{c} \text{p#} \\
\text{p1} \\
\text{p2} \end{array} = \begin{array}{c} \text{BAD_S} \\
\text{s#} \\
\text{s1} \end{array} \]
Division

- Observations: ~reverse of cartesian product
- It can be derived from the 5 fundamental operators ( (!! )
- How?
Division

- Answer:

\[ r \div S = \pi_{(R-S)}(r) - \pi_{(R-S)}[(\pi_{(R-S)}(r) \times S) - r] \]

- Observation: find ‘good’ suppliers, and subtract! (double negation)
### Division

**Answer:**

\[ r \div S = \pi_{(R-S)}(r) - \pi_{(R-S)}[(\pi_{(R-S)}(r) \times S) - r] \]

*R: attributes of r
S: attributes of s*

**Observation:** find ‘good’ suppliers, and subtract! (double negation)
Division

Answer:

\[ r \div S = \pi_{(R-S)}(r) - \pi_{(R-S)}\left[(\pi_{(R-S)}(r) \times S) - r\right] \]

<table>
<thead>
<tr>
<th>SHIPMENT</th>
<th>(s^#)</th>
<th>(p^#)</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>p1</td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>p1</td>
<td></td>
</tr>
<tr>
<td>s1</td>
<td>p2</td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td>p1</td>
<td></td>
</tr>
<tr>
<td>s5</td>
<td>p3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ABOMB</th>
<th>(p^#)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>p2</td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{BAD}_S \]

\[ \text{BAD}_S \]

All suppliers

All bad parts
Division

Answer:

\[ r \div S = \pi_{(R-S)}(r) - \pi_{(R-S)}[(\pi_{(R-S)}(r) \times S) - r] \]

all possible suspicious shipments
Division

Answer:

\[ r \div S = \pi_{(R-S)}(r) - \pi_{(R-S)}[(\pi_{(R-S)}(r) \times S) - r] \]

**SHIPMENT**

<table>
<thead>
<tr>
<th>s#</th>
<th>p#</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>p1</td>
</tr>
<tr>
<td>s2</td>
<td>p1</td>
</tr>
<tr>
<td>s1</td>
<td>p2</td>
</tr>
<tr>
<td>s3</td>
<td>p1</td>
</tr>
<tr>
<td>s5</td>
<td>p3</td>
</tr>
</tbody>
</table>

**ABOMB**

<table>
<thead>
<tr>
<th>p#</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
</tr>
<tr>
<td>p2</td>
</tr>
</tbody>
</table>

**BAD_S**

<table>
<thead>
<tr>
<th>s#</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
</tr>
</tbody>
</table>

All possible suspicious shipments that didn’t happen
Division

Answer:

\[ r \div S = \pi_{(R-S)}(r) - \pi_{(R-S)}[(\pi_{(R-S)}(r) \times s) - r] \]

- **SHIPMENT**
<table>
<thead>
<tr>
<th>s#</th>
<th>p#</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>p1</td>
</tr>
<tr>
<td>s2</td>
<td>p1</td>
</tr>
<tr>
<td>s1</td>
<td>p2</td>
</tr>
<tr>
<td>s3</td>
<td>p1</td>
</tr>
<tr>
<td>s5</td>
<td>p3</td>
</tr>
</tbody>
</table>

- **ABOMB**
<table>
<thead>
<tr>
<th>p#</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
</tr>
<tr>
<td>p2</td>
</tr>
</tbody>
</table>

- **BAD_S**
<table>
<thead>
<tr>
<th>s#</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
</tr>
</tbody>
</table>

all suppliers who missed at least one suspicious shipment, i.e.: ‘good’ suppliers
Quick Quiz: Independence of Operators

\[ R \cap S = R - (R - S) \]
\[ R \Join_C = \sigma_C (R \times S) \]
\[ R \Join S = ?? \]
Quick Quiz: Independence of Operators

\[ R \bowtie S \]

- Suppose R and S share the attributes A₁,A₂,..Aₙ
- Let L be the list of attributes in \( R \setminus \cup \) list of attributes in S (so no duplicate attributes)
- Let C be the condition

\[ R.A₁ = S.A₁ \And R.A₂ = S.A₂ \And \ldots \And R.Aₙ = S.Aₙ \]

\[ R \bowtie S = \pi_L(\sigma_C(R \times S)) \]
Linear Notation for Relational Algebra

- Relational algebra expressions can become very long.

- Use linear notation to store results of intermediate expressions.
  - A relation name and a parenthesized list of attributes for that relation
  - Use Answer as the conventional name for the final result
  - The assignment symbol :=
  - Any expression in relational algebra on the right
Example of Linear Notation

- Name pairs of students who live at the same address.
- Normal expression:

\[
\pi S1.Name,S2.Name \\
\sigma S1.Address=S2.Address \\
(\rho_{S1}(Students) \times \rho_{S2}(Students))
\]
Example of Linear Notation

- Normal expression:

\[
\pi_{S_1.Name, S_2.Name} (\sigma_{S_1.Address = S_2.Address} (\rho_{S_1}(Students) \times \rho_{S_2}(Students)))
\]

- Linear Notation:

Pairs(P1, N1, A1, P2, N2, A2) := \rho_{S_1}(Students) \times \rho_{S_2}(Students)
Matched(P1, N1, A1, P2, N2, A2) := \sigma_{A_1 = A_2}(Pairs(P1, N1, A1, P2, N2, A2))
Answer(Name1, Name2) := \pi_{N_1, N_2}(Matched(P1, N1, A1, P2, N2, A2))
Interpreting Queries Involving Multiple Relations

- SELECT A, B FROM R, S WHERE C;
- Nested loops:
  for each tuple t1 in R
    for each tuple t2 in S
      if the attributes in t1 and t2 satisfy C
        output the tuples involving attributes A and B
Interpreting Queries Involving Multiple Relations

- SELECT A, B FROM R, S WHERE C;
- Conversion to relational algebra:

\[ \pi_{A,B}(\sigma_C(R \times S)) \]

Compute R X S

Apply selection operator \(\sigma()\) to R X S

Project the result tuples to attributes A and B
Aggregate functions

- find avg grade, across all students
  
  select ??
  
  from takes

```
<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>15-413</td>
<td>4</td>
</tr>
<tr>
<td>234</td>
<td>15-413</td>
<td>3</td>
</tr>
</tbody>
</table>
```
Aggregate functions

- find avg grade, across all students
  select avg(grade)
  from takes

- result: a single number
- Which other functions?
Aggregate Operators

- **COUNT (*)**
- **COUNT ( [DISTINCT] A)**
  - A is a column
- **SUM ( [DISTINCT] A)**
- **AVG ( [DISTINCT] A)**
- **MAX (A)**
- **MIN (A)**
Aggregate functions

- find total number of enrollments
  
  select count(*)

  from takes

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>15-413</td>
<td>4</td>
</tr>
<tr>
<td>234</td>
<td>15-413</td>
<td>3</td>
</tr>
</tbody>
</table>
Aggregate functions

- find total number of students in 15-413
  select count(*)
  from takes
  where c-id = "15-413"

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>15-413</td>
<td>4</td>
</tr>
<tr>
<td>234</td>
<td>15-413</td>
<td>3</td>
</tr>
</tbody>
</table>
Find name and age of the oldest sailor(s)

### Sailors

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>age</th>
<th>ratings</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>Tom</td>
<td>34</td>
<td>5.0</td>
</tr>
</tbody>
</table>

```
SELECT S.sname, MAX (S.age) 
FROM  Sailors S
```

- This is illegal, but why?
  - Cannot combine a column with a value

```
SELECT S.sname, S.age 
FROM  Sailors S 
WHERE  S.age = (SELECT MAX (S2.age) FROM  Sailors S2)
```
GROUP BY and HAVING

- So far, aggregate operators are applied to all (qualifying) tuples.
  - Can we apply them to each of several groups of tuples?
- Example: find the age of the youngest sailor for each rating level.
  - In general, we don’t know how many rating levels exist, and what the rating values for these levels are!
  - Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this:

\[
\text{For } i = 1, 2, \ldots, 10: \quad \text{SELECT} \quad \text{MIN} \quad (S.\text{age}) \\
\text{FROM} \quad \text{Sailors} \quad S \\\n\text{WHERE} \quad S.\text{rating} = i
\]
Find the age of the youngest sailor for each rating level

SELECT S.rating, MIN(S.age) as age
FROM Sailors S
GROUP BY S.rating

(1) The sailors tuples are put into “same rating” groups.

(2) Compute the Minimum age for each rating group.

<table>
<thead>
<tr>
<th>Sid</th>
<th>Sname</th>
<th>Rating</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>85</td>
<td>Art</td>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>32</td>
<td>Andy</td>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>3</td>
<td>63.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rating</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
</tbody>
</table>
Find the age of the youngest sailor for each rating level that has at least 2 members

```sql
SELECT S.rating, MIN (S.age) as minage
FROM Sailors S
GROUP BY S.rating
HAVING COUNT(*) > 1
```

1. The sailors tuples are put into “same rating” groups.
2. Eliminate groups that have < 2 members.
3. Compute the Minimum age for each rating group.
### Drill

- find total number of students in each course

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>15-413</td>
<td>4</td>
</tr>
<tr>
<td>234</td>
<td>15-413</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c-id</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-413</td>
<td>2</td>
</tr>
</tbody>
</table>
Drill

- find total number of students in each course
  
  select c-id, count(*)
  
  from takes
  
  group by c-id
  
  order by c-id

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>15-413</td>
<td>4</td>
</tr>
<tr>
<td>234</td>
<td>15-413</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c-id</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-413</td>
<td>2</td>
</tr>
</tbody>
</table>
Drill

- find total number of students in each course, and sort by count, decreasing

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>15-413</td>
<td>4</td>
</tr>
<tr>
<td>234</td>
<td>15-413</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c-id</th>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-413</td>
<td>2</td>
</tr>
</tbody>
</table>
Drill

- find total number of students in each course, and sort by count, decreasing
  
  select c-id, count(*) as pop
  
  from takes
  
  group by c-id
  
  order by pop desc

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>15-413</td>
<td>4</td>
</tr>
<tr>
<td>234</td>
<td>15-413</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c-id</th>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-413</td>
<td>2</td>
</tr>
</tbody>
</table>
Queries With \textit{GROUP BY} and \textit{HAVING}

\begin{itemize}
\setlength\itemsep{0em}
\item The \textit{target-list} contains (i) attribute names (ii) terms with aggregate operations (e.g., $\text{AVG} (S.\text{age})$).
\item The attribute list (e.g., $S.\text{rating}$) in \textit{target-list} must be in \textit{grouping-list}.
\item The attributes in group-qualification must be in \textit{grouping-list}.
\end{itemize}

\begin{align*}
\text{SELECT} & \quad [\text{DISTINCT}] \text{ target-list} \\
\text{FROM} & \quad \text{relation-list} \\
\text{WHERE} & \quad \text{qualification} \\
\text{GROUP BY} & \quad \text{grouping-list} \\
\text{HAVING} & \quad \text{group-qualification}
\end{align*}

\begin{align*}
\text{SELECT} & \quad S.\text{rating}, \text{MIN} (S.\text{age}) \text{ as age} \\
\text{FROM} & \quad \text{Sailors} S \\
\text{GROUP BY} & \quad S.\text{rating} \\
\text{HAVING} & \quad S.\text{rating} > 5
\end{align*}
Motivation for Subqueries

- Find the name of the professor who teaches “CS 4604.”

```
SELECT Name
FROM Professors, Teach
WHERE (PID = ProfessorPID) AND (Number = '4604') AND (DeptName = 'CS');
```

- Do we need to take the natural join of two big relations just to get a relation with one tuple?

- Can we rewrite the query without using a join?
Nesting

- A query can be put inside another query
- Most commonly in the WHERE clause
- Sometimes in the FROM clause (depending on the software)
- This subquery is executed first (if possible)