CS 4604: Introduction to Database Management Systems

B. Aditya Prakash
Lecture #3: SQL---Part 1
Last lecture

- Relational Algebra
Quick Quiz: Independence of Operators

\[ R \cap S = R - (R - S) \]
\[ R \bowtie_C = \sigma_C (R \times S) \]
\[ R \bowtie S = ?? \]
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 \Union list of attributes in S (so no duplicate attributes)
- Let C be the condition

\[ R.A1 = S.A1 \text{ AND } R.A2 = S.A2 \text{ AND } \ldots \text{ R.An} = S.An \]

\[ R \bowtie S = \pi_L(\sigma_C(R \times S)) \]
Quick Aside: RA queries can become long!

- Normal expression:

\[
\pi_{S1.Name, S2.Name}(
\sigma_{S1.Address = S2.Address}
(\rho_{S1}(Students) \times \rho_{S2}(Students)))
\]

- “Linear” Notation:

\[
\text{Pairs}(P1, N1, A1, P2, N2, A2) := \rho_{S1}(Students) \times \rho_{S2}(Students)
\]
\[
\text{Matched}(P1, N1, A1, P2, N2, A2) := \sigma_{A1 = A2}(\text{Pairs}(P1, N1, A1, P2, N2, A2))
\]
\[
\text{Answer}(Name1, Name2) := \pi_{N1, N2}(\text{Matched}(P1, N1, A1, P2, N2, A2))
\]
This lecture

- Structured Query Language (SQL)
  - Pronounced ‘Sequel’
Overview - detailed - SQL

- **DML**
  - select, from, where, renaming
  - set operations
  - ordering
  - aggregate functions
  - nested subqueries

- other parts: DDL, constraints etc.
Relational Query Languages

- A major strength of the relational model: supports simple, powerful *querying* of data.
- Two sublanguages:
  - DDL – Data Definition Language
    - define and modify schema (at all 3 levels)
  - DML – Data Manipulation Language
    - Queries can be written intuitively.
The DBMS is responsible for efficient evaluation.

- Query optimizer: re-orders operations and generates query plan
The SQL Query Language

- The most widely used relational query language.
  - Major standard is SQL-1999 (=SQL3)
    - Introduced “Object-Relational” concepts
    - SQL 2003, SQL 2008 have small extensions
  - SQL92 is a basic subset
SQL (cont’d)

– PostgreSQL has some “unique” aspects (as do most systems).
– XML is the next challenge for SQL.
- Most popular embedded db in the world
  - Iphone (iOS), Android, Chrome....
- (Very) Easy to use: no need to set it up
- Self-contained: data+schema
- DB on your laptop: useful for testing, understanding....
DML

General form

```
select a1, a2, ... an
from r1, r2, ... rm
where P
[order by ....]
[group by ...]
[having ...]
```
Reminder: mini-U db

<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>CLASS</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c-id</td>
<td>c-name</td>
<td>units</td>
</tr>
<tr>
<td>4602</td>
<td>s.e.</td>
<td>2</td>
</tr>
<tr>
<td>4603</td>
<td>o.s.</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TAKES</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SSN</td>
<td>c-id</td>
<td>grade</td>
</tr>
<tr>
<td>123</td>
<td>4613</td>
<td>A</td>
</tr>
<tr>
<td>234</td>
<td>4613</td>
<td>B</td>
</tr>
</tbody>
</table>
DML - eg:

find the ssn(s) of everybody called “smith”

```
select ssn
from student
where name="smith"
```
DML - observation

General form

\[
\text{select } a_1, a_2, \ldots \text{ an} \\
\text{from } r_1, r_2, \ldots \text{ rm} \\
\text{where } P
\]

equivalent rel. algebra query?
DML - observation

General form

```
select a1, a2, ... an
from r1, r2, ... rm
where P
```

\[ \pi_{a_1,a_2,...,a_n} (\sigma_P (r_1 \times r_2 \times ... \times r_m)) \]
DML – observation – Set VS Bags

General form

```
select distinct a1, a2, ... an
from r1, r2, ... rm
where P

\pi_{a1,a2,...an}(\sigma_P(r1 \times r2 \times ... \times rm))
```

NOTE:

- Relational Algebra is **set semantics** (everything is a set), so removes duplicates automatically.

- SQL is **bag semantics** (everything is a multiset), so removes duplicates only when asked to (using distinct)
select clause

```sql
select [distinct | all ] name
from student
where address="main"
```
find ssn(s) of all “smith”’s on “main”

select ssn
from student
where address=“main” and
  name = “smith”
where clause

- boolean operators (and or not …)
- comparison operators (<, >, =, …)
- and more…
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 ssns 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
from clause

find names of people taking 4604

<table>
<thead>
<tr>
<th>STUDENT</th>
<th>CLASS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ssn</td>
<td>Name</td>
</tr>
<tr>
<td>123</td>
<td>smith</td>
</tr>
<tr>
<td>234</td>
<td>jones</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TAKES</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSN</td>
</tr>
<tr>
<td>123</td>
</tr>
<tr>
<td>234</td>
</tr>
</tbody>
</table>
from clause

find names of people taking 4604

select name

from student, takes

where ???
from clause

find names of people taking 4604

select name
from student, takes
where student.ssn = takes.ssn and
takes.c-id = "4604"
renaming - tuple variables

find names of people taking 4604

select name
from ourVeryOwnStudent, studentTakingClasses
where ourVeryOwnStudent.ssn =
    studentTakingClasses.ssn
and studentTakingClasses.c-id = “4604”
renaming - tuple variables

find names of people taking 4604

```sql
select name
from ourVeryOwnStudent as S,
    studentTakingClasses as T
where S.ssn = T.ssn
    and T.c-id = "4604"
```
renaming - self-join

- self-joins: find Tom’s grandparent(s)

<table>
<thead>
<tr>
<th>PC</th>
<th>p-id</th>
<th>c-id</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mary</td>
<td>Tom</td>
</tr>
<tr>
<td></td>
<td>Peter</td>
<td>Mary</td>
</tr>
<tr>
<td></td>
<td>John</td>
<td>Tom</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PC</th>
<th>p-id</th>
<th>c-id</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mary</td>
<td>Tom</td>
</tr>
<tr>
<td></td>
<td>Peter</td>
<td>Mary</td>
</tr>
<tr>
<td></td>
<td>John</td>
<td>Tom</td>
</tr>
</tbody>
</table>
renaming - self-join

find grandparents of “Tom” (PC(p-id, c-id))

```
select gp.p-id
from PC as gp, PC
where gp.c-id = PC.p-id
  and PC.c-id = “Tom”
```
renaming - theta join

find course names with more units than 4604

```sql
select c1.c-name
from class as c1, class as c2
where c1.units > c2.units
  and c2.c-id = "4604"
```
renaming - theta join

find course names with more units than 4604

```
select c1.c-name
from class as c1, class as c2
where c1.units > c2.units
  and c2.c-id = "4604"
```
Overview - detailed - SQL

- DML
  - select, from, where
  - set operations
  - ordering
  - aggregate functions
  - nested subqueries

- other parts: DDL, constraints etc.
set operations

find ssn of people taking both 4604 and 4613

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>4613</td>
<td>A</td>
</tr>
<tr>
<td>234</td>
<td>4613</td>
<td>B</td>
</tr>
</tbody>
</table>
set operations

find ssn of people taking both 4604 and 4613

```sql
select ssn
from takes
where c-id=“4604” and c-id=“4613”
```
set operations

find ssn of people taking both 4604 and 4613

\[
\text{(select ssn from takes where c-id=“4604” )}
\]

Intersect

\[
\text{(select ssn from takes where c-id=“4613” )}
\]

other ops: union , except
Overview - detailed - SQL

- DML
  - select, from, where
  - set operations
  - ordering
  - aggregate functions
  - nested subqueries

- other parts: DDL, constraints etc.
Ordering

find student records, sorted in name order

select *
from student
where
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

```sql
select *
from student
order by name, ssn desc
```
Overview - detailed - SQL

- DML
  - select, from, where
  - set operations
  - ordering
  - aggregate functions
  - nested subqueries

- other parts: DDL, constraints etc.
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>4613</td>
<td>A</td>
</tr>
<tr>
<td>234</td>
<td>4613</td>
<td>B</td>
</tr>
</tbody>
</table>
find avg grade, across all students

select \text{avg}(\text{grade})
from takes

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

- $A$: sum count min max (std)
Aggregate functions

find total number of enrollments

```sql
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>4613</td>
<td>A</td>
</tr>
<tr>
<td>234</td>
<td>4613</td>
<td>B</td>
</tr>
</tbody>
</table>
find total number of students in 4604

```sql
select count(*)
from takes
where c-id='4604'
```

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>4613</td>
<td>A</td>
</tr>
<tr>
<td>234</td>
<td>4613</td>
<td>B</td>
</tr>
</tbody>
</table>
Aggregate functions

find total number of students in each course

select count(*)
from takes
where ???

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>4613</td>
<td>A</td>
</tr>
<tr>
<td>234</td>
<td>4613</td>
<td>B</td>
</tr>
</tbody>
</table>
Aggregate functions

find total number of students in each course

select c-id, count(*)
from takes
group by c-id

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>4613</td>
<td>A</td>
</tr>
<tr>
<td>234</td>
<td>4613</td>
<td>B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c-id</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>4613</td>
<td>2</td>
</tr>
</tbody>
</table>
Aggregate functions

find total number of students in each course

```sql
select c-id, count(*)
from takes
group by c-id
order by c-id
```

### TAKES

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>4613</td>
<td>A</td>
</tr>
<tr>
<td>234</td>
<td>4613</td>
<td>B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c-id</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>4613</td>
<td>2</td>
</tr>
</tbody>
</table>
Aggregate functions

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

TAKES

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>4613</td>
<td>A</td>
</tr>
<tr>
<td>234</td>
<td>4613</td>
<td>B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c-id</th>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>4613</td>
<td>2</td>
</tr>
</tbody>
</table>
Aggregate functions- ‘having’

find students with GPA > 3.0

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>4613</td>
<td>4</td>
</tr>
<tr>
<td>234</td>
<td>4613</td>
<td>3</td>
</tr>
</tbody>
</table>
Aggregate functions- ‘having’

find students with GPA > 3.0

select ???, avg(grade)
from takes

group by ???

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>4613</td>
<td>4</td>
</tr>
<tr>
<td>234</td>
<td>4613</td>
<td>3</td>
</tr>
</tbody>
</table>
Aggregate functions- ‘having’

find students with GPA > 3.0

```
select ssn, avg(grade)
from takes
group by ssn
???
```
Aggregate functions- ‘having’

find students with GPA > 3.0

select ssn, \text{avg}(\text{grade})

from takes

group by ssn

having \text{avg}(\text{grade}) > 3.0

‘having’ <-> ‘where’ for groups

\begin{tabular}{|c|c|c|}
\hline
SSN & c-id & grade \\
\hline
123 & 4613 & 4 \\
234 & 4613 & 3 \\
\hline
\end{tabular}

\begin{tabular}{|c|c|}
\hline
SSN & \text{avg}(\text{grade}) \\
\hline
123 & 4 \\
234 & 3 \\
\hline
\end{tabular}
Aggregate functions- ‘having’

find students and GPA, for students with > 5 courses

select ssn, avg(grade)
from takes

group by ssn
having count(*) > 5
Drill: Find the age of the youngest sailor for each rating level

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

(2) Compute the Minimum age for each rating group.
Drill: 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>
Drill: Find the age of the youngest sailor for each rating level that has at least 2 members

<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>

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.

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

```
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.

<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>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
</tbody>
</table>
Overview - detailed - SQL

- DML
  - select, from, where
  - set operations
  - ordering
  - aggregate functions
  - nested subqueries

- other parts: DDL, constraints etc.
Recap: DML

General form

```
select a1, a2, ... an  
from r1, r2, ... rm  
where P
[order by ....]  
[group by ...]  
[having ...]
```
DML - nested subqueries

find names of students of 4604

```sql
select name
from student
where ... 
```

“ssn in the set of people that take 4604”
DML - nested subqueries

find names of students of 15-415

```sql
select name
from student
where ............
  select ssn
  from takes
where c-id ="4604"
```
find names of students of 15-415

    select name
    from student
    where ssn in (  
      select ssn  
      from takes  
      where c-id ="4604"
    )
DML - nested subqueries

- ‘in’ compares a value with a set of values
- ‘in’ can be combined other boolean ops
- it is redundant (but user friendly!):
  
  ```
  select name
  from student ..... 
  where c-id = "4604" ..... 
  ```
DML - nested subqueries

- ‘in’ compares a value with a set of values
- ‘in’ can be combined other boolean ops
- it is redundant (but user friendly!):

```sql
select name
from student, takes
where c-id = "4604" and
    student.ssn = takes.ssn
```
DML - nested subqueries

find names of students taking 4604 and living on “main str”

select name
from student
where address=“main str” and ssn in
  ( select ssn from takes where c-id =“4604”)

DML - nested subqueries

- ‘in’ compares a value with a set of values
- other operators like ‘in’
find student record with highest ssn

```
select *
from student
where ssn
    is greater than every other ssn
```
DML - nested subqueries

find student record with highest ssn

select *
from student
where ssn greater than every
  select ssn from student
find student record with highest ssn

select *
from student
where ssn > all (
    select ssn from student)
find student record with highest ssn

select *
from student
where ssn >= all ( select ssn from student)
DML - nested subqueries

find student record with highest ssn - without nested subqueries?

```sql
select S1.ssn, S1.name, S1.address
from student as S1, student as S2
where S1.ssn > S2.ssn
```

is not the answer (what does it give?)
## DML - nested subqueries

<table>
<thead>
<tr>
<th>STUDENT</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ssn</strong></td>
<td><strong>Name</strong></td>
<td><strong>Address</strong></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>STUDENT</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ssn</strong></td>
<td><strong>Name</strong></td>
<td><strong>Address</strong></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>

\[
S1 \times S2
\]

\[
S1. ssn > S2. ssn
\]
DML - nested subqueries

```sql
select S1.ssn, S1.name, S1.address
from student as S1, student as S2
where S1.ssn > S2.ssn
```

gives all but the smallest ssn -

aha!
find student record with highest ssn - without nested subqueries?

```sql
select S1.ssn, S1.name, S1.address
from student as S1, student as S2
where S1.ssn < S2.ssn
```
gives all but the highest - therefore....
DML - nested subqueries

find student record with highest ssn - without nested subqueries?

\[(\text{select } * \text{ from student}) \text{ except} \ (\text{select } S1.ssn, S1.name, S1.address \text{ from student as } S1, \text{ student as } S2 \text{ where } S1.ssn < S2.ssn)\]
DML - nested subqueries

\[
\text{(select * from student) except (select S1.ssn, S1.name, S1.address from student as S1, student as S2 where S1.ssn < S2.ssn)}
\]

\[
\begin{align*}
\text{select} & \quad * \\
\text{from} & \quad \text{student} \\
\text{where} & \quad \text{ssn} \geq \text{all (select ssn from student)}
\end{align*}
\]
Drill: Even more readable than

```
select * from student
where ssn >= all (select ssn from student)
```
Drill: Even more readable than

```sql
select * from student
where ssn >= all (select ssn from student)
```

```sql
select * from student
where ssn in
(select max(ssn) from student)
```
from clause

Drill: find the ssn of the student with the highest GPA

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

<table>
<thead>
<tr>
<th>CLASS</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c-id</td>
<td>c-name</td>
<td>units</td>
</tr>
<tr>
<td>4602</td>
<td>s.e.</td>
<td>2</td>
</tr>
<tr>
<td>4603</td>
<td>o.s.</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TAKES</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SSN</td>
<td>c-id</td>
<td>grade</td>
</tr>
<tr>
<td>123</td>
<td>4613</td>
<td>A</td>
</tr>
<tr>
<td>234</td>
<td>4613</td>
<td>B</td>
</tr>
</tbody>
</table>
DML - nested subqueries

Drill: find the ssn and GPA of the student with the highest GPA

```
select ssn, avg(grade) from takes
where
```
DML - nested subqueries

Drill: find the ssn and GPA of the student with the highest GPA

```sql
select ssn, avg(grade) from takes
  group by ssn
having avg(grade) greater than every other GPA on file
```
DML - nested subqueries

Drill: find the ssn and GPA of the student with the highest GPA

```sql
select ssn, avg(grade) from takes
  group by ssn
having avg( grade ) >= all
  ( select avg( grade )
    from student group by ssn )
```

} all GPAs
DML - nested subqueries

- ‘in’ and ‘>= all’ compares a value with a set of values
- other operators like these?
DML - nested subqueries

- `<all(), <>all() ...`
- `<>all` is identical to `not in`
- `>some(), >= some () ...`
- `= some()` is identical to `in`
- `exists`
DML - nested subqueries

Drill for ‘exists’ : find all courses that nobody enrolled in

select c-id from class ....with no tuples in ‘takes’
DML - nested subqueries

Drill for ‘exists’: find all courses that nobody enrolled in

\[
\text{select c-id from class}
\]
\[
\text{where not exists}
\]
\[
(\text{select * from takes}
\]
\[
\text{where class.c-id = takes.c-id})
\]
Correlated vs Uncorrelated

- The previous subqueries did not depend on anything outside the subquery
  - ...and thus need to be executed just once.
  - These are called **uncorrelated**.

- A **correlated** subquery depends on data from the outer query
  - ... and thus has to be executed for each row of the outer table(s)
Correlated Subqueries

- Find course names that have been used for two or more courses.

SELECT CourseName
FROM Courses AS First
WHERE CourseName IN
  (SELECT CourseName
   FROM Courses
   WHERE (Number <> First.Number)
   AND (DeptName <> First.DeptName)
  );
Evaluating Correlated Subqueries

```
SELECT CourseName
FROM Courses AS First
WHERE CourseName IN
    (SELECT CourseName
     FROM Courses
     WHERE (Number <> First.Number)
     AND (DeptName <> First.DeptName)
    );
```

- Evaluate query by looping over tuples of First, and for each tuple evaluate the subquery.
- Scoping rules: an attribute in a subquery belongs to one of the tuple variables in that subquery’s FROM clause, or to the immediately surrounding subquery, and so on.
Overview - detailed - SQL

- DML
  - select, from, where
  - set operations
  - ordering
  - aggregate functions
  - nested subqueries

- other parts: DDL, constraints etc.
(Next Week) Overview - detailed – SQL

- DML

- other parts:
  - views
  - modifications
  - joins
  - DDL
  - Constraints