SQL: Data Manipulation Language

csc343, Introduction to Databases
Fatemeh Nargesian (slides are borrowed from Diane Horton)
Fall 2015
GROUP BY - Example

• How many courses (offerings) has each instructor offered?

• `SELECT instructor, count(*)`
  `FROM Offerings`
  `GROUP BY instructor;`

<table>
<thead>
<tr>
<th>instructor</th>
<th>count(*)</th>
<th>..</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ricky</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Smith</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Meyer</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>o1</th>
<th>dept</th>
<th>cNum</th>
<th>term</th>
<th>instructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>csc</td>
<td>343</td>
<td>201501</td>
<td>Ricky</td>
<td></td>
</tr>
<tr>
<td>csc</td>
<td>180</td>
<td>201502</td>
<td>Rikcy</td>
<td></td>
</tr>
<tr>
<td>csc</td>
<td>180</td>
<td>201502</td>
<td>Smith</td>
<td></td>
</tr>
<tr>
<td>math</td>
<td>555</td>
<td>201501</td>
<td>Meyer</td>
<td></td>
</tr>
</tbody>
</table>
GROUP BY - Example

• How many courses (offerings) has each instructor offered?

  • SELECT instructor, count(*)
    FROM Offerings
    WHERE dept = 'csc'
    GROUP BY instructor;
Grouping

• If we follow a SELECT-FROM-WHERE expression with GROUP BY <attributes>
  • The tuples are grouped according to the values of those attributes, and
  • any aggregation is applied only within each group.
Restrictions on aggregation

• If any aggregation is used, then each element of the SELECT list must be either:
  • aggregated, or
  • an attribute on the GROUP BY list.
• Otherwise, it doesn’t even make sense to include the attribute.
HAVING Clauses

- WHERE let’s you decide which tuples to keep.
- Similarly, you can decide which groups to keep.
- Syntax:
  ```
  ...
  GROUP BY «attributes»
  HAVING «condition»
  ```
- Semantics:
  Only groups satisfying the condition are kept.
HAVING - Example

- SELECT instructor, count(*)
  FROM Offerings
  WHERE dept = 'csc'
  GROUP BY instructor
  HAVING count(*) > 2;
Requirements on HAVING clauses

• Outside subqueries, HAVING may refer to attributes only if they are either:
  • aggregated, or
  • an attribute on the GROUP BY list.

• (The same requirement as for SELECT clauses with aggregation).
# Order of execution of a SQL query

<table>
<thead>
<tr>
<th>Query order</th>
<th>Execution order</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>FROM</td>
</tr>
<tr>
<td>FROM</td>
<td>WHERE</td>
</tr>
<tr>
<td>WHERE</td>
<td>GROUP BY</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>HAVING</td>
</tr>
<tr>
<td>HAVING</td>
<td>SELECT</td>
</tr>
<tr>
<td>ORDER BY</td>
<td>ORDER BY</td>
</tr>
</tbody>
</table>
In-class Exercises!
Set operations
Tables can have duplicates in SQL

• A table can have duplicate tuples, unless this would violate an integrity constraint.
• And SELECT-FROM-WHERE statements leave duplicates in unless you say not to.
• Why?
  • Getting rid of duplicates is expensive!
  • We may want the duplicates because they tell us how many times something occurred.
Relational Algebra with Bags

- Reference: section 5.1 of the text.
- Behaviour of most operations is no different.
  - $\sigma$, $\rho$: as before
  - $\pi$: duplicates are not removed.
  - joins: duplicates can proliferate
Bags

- SQL treats tables as “bags” (or “multisets”) rather than sets.
- Bags are just like sets, but duplicates are allowed.
- \{6, 2, 7, 1, 9\} is a set (and a bag)
  \{6, 2, 2, 7, 1, 9\} is not a set, but is a bag.
- Like with sets, order doesn’t matter.
  \{6, 2, 7, 1, 9\} = \{1, 2, 6, 7, 9\}
Union, Intersection, and Difference

• These are expressed as:
  
  («subquery») UNION («subquery»)
  («subquery») INTERSECT («subquery»)
  («subquery») EXCEPT («subquery»)

• The brackets are mandatory.
• The operands must be queries; you can’t simply use a relation name.
Example

```
(SELECT sid
 FROM Took
 WHERE grade > 95)
UNION

(SELECT sid
 FROM Took
 WHERE grade < 50);
```

Schemas should match!
In-class Exercises!
Operations $\cup$, $\cap$, and $-$ with Bags

• For $\cup$, $\cap$, and $-$ the number of occurrences of a tuple in the result requires some thought.

• Exercises:
  1. $\{1, 1, 1, 3, 7, 7, 8\} \cup \{1, 5, 7, 7, 8, 8\}$
  2. $\{1, 1, 1, 3, 7, 7, 8\} \cap \{1, 5, 7, 7, 8, 8\}$
  3. $\{1, 1, 1, 3, 7, 7, 8\} - \{1, 5, 7, 7, 8, 8\}$
1. \{1, 1, 1, 3, 7, 7, 8\} \cup \{1, 5, 7, 7, 8, 8\}
   = \{1, 1, 1, 3, 7, 7, 8, 1, 5, 7, 7, 8, 8\}
   = \{1, 1, 1, 1, 3, 5, 7, 7, 7, 7, 8, 8, 8\}

2. \{1, 1, 1, 3, 7, 7, 8\} \cap \{1, 5, 7, 7, 8, 8\}
   = \{1, 7, 7, 8\}

3. \{1, 1, 1, 3, 7, 7, 8\} - \{1, 5, 7, 7, 8, 8\}
   = \{1, 1, 3\}
Operations $\cup$, $\cap$, and $-$ with Bags

- Suppose tuple $t$ occurs
  - $m$ times in relation $R$, and
  - $n$ times in relation $S$.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Number of occurrences of $t$ in result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R \cap S$</td>
<td>$\min(m, n)$</td>
</tr>
<tr>
<td>$R \cup S$</td>
<td>$m + n$</td>
</tr>
<tr>
<td>$R - S$</td>
<td>$\max(m-n, 0)$</td>
</tr>
</tbody>
</table>
Bag vs Set Semantics: which is used

- We saw that a SELECT-FROM-WHERE statement uses bag semantics by default.
  - Duplicates are kept in the result.
- The set operations use set semantics by default.
  - Duplicates are *eliminated* from the result.
Motivation: Efficiency

• When doing projection, it is easier not to eliminate duplicates.
  • Just work one tuple at a time.

• For intersection or difference, it is most efficient to sort the relations first.
  • At that point you may as well eliminate the duplicates anyway.
Controlling Duplicate Elimination

• We can force the result of a SFW query to be a set by using **SELECT DISTINCT ...**

  SELECT DISTINCT sid
  FROM Took
  WHERE grade > 95

• We can force the result of a set operation to be a bag by using **ALL**, e.g.,

  (SELECT sid
   FROM Took
   WHERE grade > 95)
  UNION ALL

  (SELECT sid
   FROM Took
   WHERE grade > 95)
Views
The idea

- A view is a relation defined in terms of stored tables (called base tables) and other views.
- Access a view like any base table.
- Two kinds of view:
  - Virtual: no tuples are stored; view is just a query for constructing the relation when needed.
  - Materialized: actually constructed and stored. Expensive to maintain!
- We’ll use only virtual views.
  - PostgreSQL did not support materialized views until version 9.3 (which we are not running).
Example: defining a virtual view

• A view for students who earned an 80 or higher in a CSC course.

```sql
CREATE VIEW topresults as
SELECT firstname, surname, cnum
FROM Student, Took, Offering
WHERE
  Student.sid = Took.sid AND
  Took.oid = Offering.oid AND
  grade >= 80 AND dept = 'CSC';
```
Uses for views

• Break down a large query.
• Provide another way of looking at the same data, e.g., for one category of user.
In-class Exercises!
GROUP BY - Example

• How many courses (offerings) has each instructor offered?
• SELECT instructor, count(*)
  FROM Offerings
  GROUP BY instructor;

| instructor | count(*) | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ricky</td>
<td>2</td>
<td>...</td>
</tr>
<tr>
<td>Smith</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>Meyer</td>
<td>1</td>
<td>...</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>oID</th>
<th>dept</th>
<th>cNum</th>
<th>term</th>
<th>instructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1</td>
<td>csc</td>
<td>343</td>
<td>201501</td>
<td>Ricky</td>
</tr>
<tr>
<td>o2</td>
<td>csc</td>
<td>180</td>
<td>201502</td>
<td>Rikcy</td>
</tr>
<tr>
<td>o3</td>
<td>csc</td>
<td>180</td>
<td>201502</td>
<td>Smith</td>
</tr>
<tr>
<td>o4</td>
<td>math</td>
<td>555</td>
<td>201501</td>
<td>Meyer</td>
</tr>
</tbody>
</table>
```
Exercise

• Write a query to find all terms when Jepson and Suzuki were both teaching. Include duplicates of the same term.
  • Offering(oID, dept, cNum, term, instructor)

• (select term from Offering where instructor = 'Suzuki')
  intersect all
  (select Term from Offering where instructor = 'Jepson');
Exercise

- Find the ID of students who have earned a grade of 85 or more in some course, or who have passed a course taught by Atwood. Use views for the intermediate steps.

- create view **High** as
  (select sid from took where grade >= 85);

- create view **HighAtwood** as
  (select sid from Took, Offering
  where Took.oid = Offering.oid and instructor = 'Atwood' and grade >= 50);

- (select * from high) union (select * from highAtwood);
Exercise

- Find all terms when csc369 was not offered.
- (select term from Offering)
  except
  (select term
   from Offering
   where dept = 'csc' and cNum = 369);
JOINS
The joins you know from RA

These can go in a FROM clause, or can be stand-alone queries:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>R, S</td>
<td>R \times S</td>
</tr>
<tr>
<td>R cross join S</td>
<td>R \times S</td>
</tr>
<tr>
<td>R natural join S</td>
<td>R \bowtie S</td>
</tr>
<tr>
<td>R join S on Condition</td>
<td>R \bowtie_{condition} S</td>
</tr>
</tbody>
</table>
Attention! Cross join!

• If relation R has m tuples and relation S has n tuples, what is the size of R cross join S?
  • \( m \times n \)

• Avoid using cross join by all means!

```sql
SELECT firstname, surname, cnum
FROM Student, Took, Offering
WHERE
  Student.sid = Took.sid AND
  Took.oid = Offering.oid AND
  grade >= 80 AND dept = 'CSC';
```
ROSI Schema

- Student(sID, surName, firstName, campus, email, cgpa)
- Course(dept, cNum, name, breadth)
- Offering(oID, dept, cNum, term, instructor)
- Took(sID, oID, grade)
In practice natural join is dangerous

• A working query can be broken by adding a column to a schema.
  • Example:
    
    ```sql
    select sID, instructor
    from Student natural join Took
    natural join Offering;
    ```
  
  • What if we add a column called `campus` to `Offering`?

• Also, having implicit comparisons impairs readability.

• Best practise: Don’t use natural join.
Example

• SELECT *
  FROM Course JOIN Offering
  ON (Course.cNum = Offering.cNum AND
  Course.dept = Offering.dept);
Dangling tuples

• With joins that require some attributes to match, tuples lacking a match are left out of the results.

• We say that they are “dangling”.

• An outer join preserves dangling tuples by padding them with \texttt{NULL} in the other relation.

• A join that doesn’t pad with \texttt{NULL} is called an inner join.
Three kinds of outer join

• **LEFT OUTER JOIN**
  • Preserves dangling tuples from the relation on the LHS by padding with nulls on the RHS.

• **RIGHT OUTER JOIN**
  • The reverse.

• **FULL OUTER JOIN**
  • Does both.
Example: joining R and S various ways

R NATURAL JOIN S

SELECT * FROM R Natural JOIN S;
Example

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

**R NATURAL FULL JOIN S**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

**R NATURAL LEFT JOIN S**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

**R NATURAL RIGHT JOIN S**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>NULL</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
Example

- SELECT *
  FROM Course INNER JOIN Offering
  ON (Course.cNum = Offering.cNum AND
  Course.dept = Offering.dept);
Summary of join expressions

- **Cartesian product**
  \[ A \times B \]  
  same as \( A, B \)

- **Theta-join**
  \[ A \Join B \text{ on } C \]  
  no padding of tuples

- **Natural join**
  \[ A \natural B \]  
  no padding of tuples

- **Left, right, full joins**
  \[ A \{\text{left}\} \Join B \]  
  padding

  \[ A \{\text{right}\} \Join B \]  
  padding

  \[ A \{\text{full}\} \Join B \]  
  padding
Keywords INNER and OUTER

• There are keywords *INNER* and *OUTER*, but you never need to use them.

• Your intentions are clear anyway:
  • You get an outer join iff you use the keywords *LEFT*, *RIGHT*, or *FULL*.
  • If you don’t use the keywords *LEFT*, *RIGHT*, or *FULL* you get an inner join.
Impact of having null values
Missing Information

• Two common scenarios:
  • Missing value. 
    E.g., we know a student has some email address, but we don’t know what it is.
  • Inapplicable attribute. 
    E.g., the value of attribute spouse for an unmarried person.
Representing missing information

- One possibility: use a special value as a placeholder. E.g.,
  - If age unknown, use 0.
  - If StNum unknown, use 999999999.
- Pros and cons?
- Better solution: use a value not in any domain. We call this a **null** value.
- Tuples in SQL relations can have **NULL** as a value for one or more components.
## NULL values

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>diane</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>will</td>
<td>20</td>
<td>8</td>
</tr>
<tr>
<td>cate</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>tom</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>micah</td>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td>jamieso</td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>
Checking for null values

- You can compare an attribute value to **NULL** with
  - **IS NULL**
  - **IS NOT NULL**
- Example:
  - `Course(dept, cNum, name, breadth)
    SELECT *
    FROM Course
    WHERE breadth IS NULL;`
In SQL we have 3 truth-values

Because of **NULL**, we need three truth-values:

- If one or both operands to a comparison is **NULL**, the comparison always evaluates to **UNKNOWN**.
- Otherwise, comparisons evaluate to **TRUE** or **FALSE**.

```
select * from R
where age < grade or grade > 5;
```
Combining truth values

• We need to know how the three truth-values combine with AND, OR and NOT.
• Can think of it in terms of the truth table.
• Or can think in terms of numbers:
  • TRUE = 1, FALSE = 0, UNKNOWN = 0.5
  • AND is min, OR is max,
  • NOT x is (1-x), i.e., it “flips” the value
Thinking of the truth-values as numbers

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>as nums</th>
<th>A and B</th>
<th>min</th>
<th>A or B</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>1, 1</td>
<td>T</td>
<td>1</td>
<td>T</td>
<td>1</td>
</tr>
<tr>
<td>TF or FT</td>
<td>1, 0</td>
<td>F</td>
<td>0</td>
<td>T</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>0, 0</td>
<td>F</td>
<td>0</td>
<td>F</td>
<td>0</td>
</tr>
<tr>
<td>TU or UT</td>
<td>1, 0.5</td>
<td>U</td>
<td>0.5</td>
<td>T</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>FU or UF</td>
<td>0, 0.5</td>
<td>F</td>
<td>0</td>
<td>U</td>
<td></td>
<td>0.5</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>0.5, 0.5</td>
<td>U</td>
<td>0.5</td>
<td>U</td>
<td>0.5</td>
</tr>
</tbody>
</table>
Impact of null values on WHERE

• A tuple is in a query result iff the WHERE clause is \textbf{TRUE}.

• \textbf{UNKNOWN} is not good enough.

• “WHERE is picky.”
Impact of null values on aggregation

• “Aggregation ignores NULL.”
• NULL never contributes to a sum, average, or count, and can never be the minimum or maximum of a column (unless every value is NULL).
• If there are no non-NUL values in a column, then the result of the aggregation is NULL.
  • Exception: COUNT of an empty set is 0.
Example

• Suppose we have a table called Person with the following content:

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>diane</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>will</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>cate</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>tom</td>
<td></td>
<td></td>
</tr>
<tr>
<td>micah</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>jamieso</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

• What is the output of this query?
  • select min(grade), max(grade), sum(grade), avg(grade), count(grade), count(*) from Person;
Example

• Suppose we have a table called Person with the following content:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>count</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>min</td>
<td>max</td>
<td>sum</td>
<td>avg</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>diane</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>will</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>cate</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>tom</td>
<td></td>
<td></td>
</tr>
<tr>
<td>micah</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>jamieso</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

• What is the output of each query below?
  • select min(age), max(age), sum(age), avg(age), count(age), count(*) from Person;
• Suppose we have a table called Person with the following content:

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>diane</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>will</td>
<td>20</td>
<td>8</td>
</tr>
<tr>
<td>cate</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>tom</td>
<td></td>
<td></td>
</tr>
<tr>
<td>micah</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>jamieso</td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>min</th>
<th>max</th>
<th>sum</th>
<th>avg</th>
<th>count</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>25</td>
<td>45</td>
<td>22.5</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

• What is the output of each query below?
  • `select min(age), max(age), sum(age), avg(age), count(age), count(*) from Person;`
Aggregation ignores nulls

<table>
<thead>
<tr>
<th></th>
<th>some nulls in A</th>
<th>All nulls in A</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>min(A)</code></td>
<td>ignore the nulls</td>
<td>null</td>
</tr>
<tr>
<td><code>max(A)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sum(A)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>avg(A)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>count(A)</code></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td><code>count(*)</code></td>
<td>all tuples count</td>
<td></td>
</tr>
</tbody>
</table>
Example-NULL and NATURAL JOIN

- We have tables R and T. Their contents are shown below.

- What is the output of each query below?
  - select * from R natural join T;
Example - NULL and Set Operations

• We have tables R and T. Their contents are shown below.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

• What is the output of each query below?
  • (SELECT * from R) INTERSECT (SELECT * from S);
Example - NULL and Set Operations

• We have tables R and T. Their contents are shown below.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

• What is the output of each query below?
  • (SELECT * from R) EXCEPT (SELECT * from S);
SQL Recap
ROSI Schema

- Student(sID, surName, firstName, campus, email, cgpa)
- Course(dept, cNum, name, breadth)
- Offering(oID, dept, cNum, term, instructor)
- Took(sID, oID, grade)
SQL Query Structure

- **SELECT** sID, AVG(grade) as stavg
  **FROM** Took t **JOIN** Offering o
  **ON** (t.oID = O.oID)
  **WHERE** term > 201401
  **GROUP BY** sID
  **HAVING** AVG(grade) > 90
  **ORDER BY** AVG(grade) **DESC**

*Avoid using cartesian product!*

*Order of execution?*
Set and Bag Operations

- Always two queries with the same result schema
- The queries are separated with brackets.
- \texttt{union}, \texttt{intersect} and \texttt{except} are set operations.
- \texttt{union all}, \texttt{intersect all} and \texttt{except all} are bag operations
- \texttt{(select term from Offering where instructor = 'Suzuki')} \texttt{intersect all} \texttt{(select term from Offering where instructor = 'Jepson')}
Views

• create view High as
  (select sid from took where grade >= 85);
• create view HighAtwood as
  (select sid
   from Took, Offering
   where Took.oid = Offering.oid and instructor = ’Atwood’ and grade >= 50);
• (select * from high) union (select * from highAtwood);
• A view is a relation defined in terms of stored tables
• It can be used as a table in other relations.
NULL Values

• Null values are used to represent missing information

• Example:
  ```sql
  SELECT *
  FROM Course
  WHERE breadth IS NULL;
  ```

• NULL values are not considered in matching process of natural join.

• NULL values are not considered in aggregation functions unless all values in a column are NULL.
Subqueries
Subqueries in a FROM clause

```
SELECT sid, dept||cnum
FROM Took, Offering
WHERE
    Took.oid = Offering.oid AND
    grade > 98;
```
Example:

• What does this do?

```
SELECT instructor, dept||cnum as course
FROM Offering,
    ( SELECT oid
        FROM took
        WHERE grade > 98  ) High
WHERE Offering.oid = High.oid;
```

• In place of a relation name in the FROM clause, we can use a subquery.

• The subquery must be parenthesized.

• Must name the result, so you can refer to it in the outer query.
Example

• What does this do?
  
  SELECT sid, dept||cnum as course, grade
  FROM Took,
  (SELECT *
   FROM Offering
   WHERE instructor='Horton') Hoffering
  WHERE Took.oid = Hoffering.oid;

• This FROM is analogous to:
  
  Took × ρ_Hoffering («subquery»)
Subquery as a value in a WHERE

• If a subquery is guaranteed to produce exactly one tuple, then the subquery can be used as a value.
• Simplest situation: that one tuple has only one component.
Example

- Find all students with a cgpa greater than that of student 99999.
  
  ```sql
  SELECT sid, surname
  FROM Student
  WHERE cgpa > (SELECT cgpa
                 FROM Student
                 WHERE sid = 99999);
  ```

- This is analogous to something we can’t do in RA:
  
  $$
  \pi_{sid, surname} \sigma_{cgpa > (\text{subquery})}
  $$
Special cases

- What if the subquery returns \texttt{NULL}?
  - evaluated to \texttt{unknown}, tuple not returned!
- What if the subquery could return more than one value?
- When a subquery can return multiple values, we can make comparisons using a quantifier:
  - \texttt{cgpa} > all of them (\texttt{ALL})
  - \texttt{cgpa} > at least one of them (\texttt{ANY})
The Operator ANY

• Syntax:
  \[ x \ «\text{comparison}\» \text{ANY («subquery»)} \]
  or equivalently
  \[ x \ «\text{comparison}\» \text{SOME («subquery»)} \]

• Semantics:
  Its value is true iff the comparison holds for at least one tuple in the subquery result, i.e.,
  \[ \exists \ y \in \ «\text{subquery results}\» \mid x \ «\text{comparison}\» y \]

• \[ x \] can be a list of attributes, but this feature is not supported by psql.
The Operator ALL

• Syntax:
  \[ x \ «\text{comparison}\» \text{ALL} («\text{subquery}\») \]

• Semantics:
  Its value is true iff the comparison holds for every tuple in the subquery result, i.e.,
  \[ \forall y \in \«\text{subquery results}\» \mid x \ «\text{comparison}\» y \]

• \(x\) can be a list of attributes, but this feature is not supported by psql.
The Operator IN

• Syntax:
  \[ x \text{ IN } (\text{«subquery»}) \]

• Semantics:
  Its value is true iff \( x \) equals at least one of the tuples in the subquery result.

• \( x \) can be a list of attributes, and psql does support this feature.
ROSI Schema

- Student(sID, surName, firstName, campus, email, cgpa)
- Course(dept, cNum, name, breadth)
- Offering(oID, dept, cNum, term, instructor)
- Took(sID, oID, grade)
Example

What does this do?

```
SELECT sid, dept||cnum AS course, grade
FROM Took NATURAL JOIN Offering
WHERE
  grade >= 80 AND
  (cnum, dept) IN (  
    SELECT cnum, dept  
    FROM Took NATURAL JOIN Offering  
    NATURAL JOIN Student  
    WHERE surname = 'Lakemeyer');
```
Exercise

Suppose we have tables $R(a, b)$ and $S(b, c)$.

1. What does this query do?
   
   ```sql
   SELECT a
   FROM R
   WHERE b IN (SELECT b FROM S);
   ```

   2. Can we express this query without using IN?
   
   ```sql
   SELECT a FROM R, S WHERE R.b = S.b;
   ```

   If a tuple from $R$ connects successfully with more than one tuple from $S$, this new query will yield duplicates that the original did not!
Announcements

• Midterm
  • Wednesday, Oct 28th, 1-2pm, location: EX100.
  • Please be there 10 minutes before the exam.
  • We will continue with the lecture 2:10-3 in MC254.
  • Office hours: Monday, Oct 26th, 2:30-3:30pm, BA7200
The Operator EXISTS

- **Syntax:**
  
  `EXISTS («subquery»)`

- **WHERE clause condition**

- **Semantics:**
  Its value is true iff the subquery has at least one tuple.
Example: NOT EXISTS

What does this do?

```sql
SELECT instructor
FROM Offering Off1
WHERE NOT EXISTS (  
  SELECT *
  FROM Offering
  WHERE
  oid <> Off1.oid AND
  Instructor = Off1.instructor
)
```

Instructors who have exactly one offering!

---

**Instructor**

- **o1**: CSC 343 Horton  
- **o2**: CSC 443 Miller  
- **o3**: MATH 101 Cooper

**Offering**

- **o1**: CSC 343  
- **o2**: CSC 443  
- **o3**: MATH 101
Queries are evaluated from the inside out.

If a name might refer to more than one thing, use the most closely nested one.

If a subquery refers only to names defined inside it, it can be evaluated once and used repeatedly in the outer query.

If it refers to any name defined outside of itself, it must be evaluated once for each tuple in the outer query. These are called correlated subqueries.
Renaming can make scope explicit

SELECT instructor
FROM Offering Off1
WHERE NOT EXISTS ( SELECT * 
    FROM Offering Off2
    WHERE
        Off2.oid <> Off1.oid AND
        Off2.instructor = Off1.instructor );
Summary: where subqueries can go

• As a relation in a FROM clause.
• As a value in a WHERE clause.
• With ANY, ALL, IN or EXISTS in a WHERE clause.
• As operands to UNION, INTERSECT or EXCEPT.
• Reference: textbook, section 6.3.
Exercise

• For each course find the instructor who has taught the most offerings of it. If there are ties, include them all. Report the course (eg “csc343”), instructor and the number of offerings of the course by that instructor.

Step 1: How many offerings of each course each instructor has taught?

CREATE VIEW Counts as
SELECT cnum || dept as course, instructor, count(oid)
FROM Offering
GROUP BY cnum, dept, instructor;
Exercise 6

CREATE VIEW Counts as
SELECT cnum || dept as course, instructor, count(oid)
FROM Offering
GROUP BY cnum, dept, instructor;

Step 2: Find instructors with maximum number of offerings in a course.

SELECT course, instructor, count
FROM Counts C1
WHERE count >= ALL (
SELECT count
FROM Counts C2
WHERE C1.course = C2.course)
ORDER BY C1.course;
Exercise 6

• Another solution:

```sql
SELECT course, instructor, count
FROM Counts C1
WHERE count = (
    SELECT max(count)
    FROM Counts C2
    WHERE C1.course = C2.course
)
ORDER BY C1.course;
```
Exercise 7

• Let’s say that a course has level “junior” if its cNum is between 100 and 299 inclusive, and has level “senior” if its cNum is between 300 and 499 inclusive. Report the average grade, across all departments and course offerings, for all junior courses and for all senior courses.

• Step 1: Find all grades of all offered courses.

```
CREATE VIEW Grades AS
    SELECT cnum, dept, grade
    FROM Offering natural join Took;
```
Exercise 7

• Step 2: Find average grade for juniors and seniors.

```
(SELECT 'junior' AS level, avg(grade) AS levelavg
    FROM Grades
    WHERE cnum >= 100 AND cnum <= 299) union

(SELECT 'senior' AS level, avg(grade) AS levelavg
    FROM Grades
    WHERE cnum >= 300 AND cnum <= 499);
```