SQL:
Data Manipulation Language

CSC343, Introduction to Databases
Daniela Rosu and Sina Meraji

Fall 2018
Introduction

• So far, we have defined database schemas and queries mathematically.
• SQL is a formal language for doing so with a DBMS.
• “Structured Query Language”, but it’s for more than writing queries.
• Two sub-parts:
  • DDL (Data Definition Language), for defining schemas.
  • DML (Data Manipulation Language), for writing queries and modifying the database.
PostgreSQL

• We’ll be working in PostgreSQL, an open-source relational DBMS.

• Learn your way around the documentation; it will be very helpful.

• Standards?
  • There are several, the most recent being SQL:2008.
  • The standards are not freely available. Must purchase from the International Standards Organization (ISO).
  • PostgreSQL supports most of it SQL:2008.
  • DBMSs vary in the details around the edges, making portability difficult.
A high-level language

• SQL is a very high-level language.
  • Say “what” rather than “how.”

• You write queries without manipulating data.
  Contrast languages like Java or C++.

• Provides physical “data independence:”
  • Details of how the data is stored can change with no impact on your queries.

• You can focus on readability.
  • But because the DMBS optimizes your query, you get efficiency.
Heads up: SELECT vs $\sigma$

- In SQL,
  - “SELECT” is for choosing columns, *i.e.*, $\Pi$.
  - Example:
    
    ```sql
    SELECT surName
    FROM Student
    WHERE campus = 'StG';
    ```

- In relational algebra,
  - “select” means choosing rows, *i.e.*, $\sigma$. 
Meaning of a query with one relation

```sql
SELECT name
FROM Course
WHERE dept = 'CSC';
```

\[ \pi_{name} (\sigma_{dept='csc'} (Course)) \]
Meaning of a query with one relation

```
SELECT name
FROM Course
WHERE dept = 'CSC';
```
... and with multiple relations

SELECT name
FROM Offering, Took
WHERE Offering.id = Took.oid and
    dept = 'CSC';

\[ \pi_{\text{name}} (\sigma \text{Offering.id=Took.id} \land \text{dept='csc'} (\text{Offering} \times \text{Took})) \]
Temporarily renaming a table

• You can rename tables (just for the duration of the statement):

```sql
SELECT e.name, d.name
FROM employee e, department d
WHERE d.name = 'marketing'
AND e.name = 'Horton';
```

• Can be convenient vs the longer full names:

```sql
SELECT employee.name, department.name
FROM employee, department
WHERE department.name = 'marketing'
AND employee.name = 'Horton';
```

• This is like ρ in relational algebra.
Self-joins

• As we know, renaming is required for self-joins.

• Example:

```sql
SELECT e1.name, e2.name
FROM employee e1, employee e2
WHERE e1.salary < e2.salary;
```
In SELECT clauses

• A * in the SELECT clause means “all attributes of this relation.”

• Example:

```sql
SELECT *
FROM Course
WHERE dept = 'CSC';
```
Renaming attributes

• Use `AS «new name»` to rename an attribute in the result.

• Example:

```sql
SELECT name AS title, dept
FROM Course
WHERE breadth;
```
Complex Conditions in a WHERE

- We can build boolean expressions with operators that produce boolean results.
  - comparison operators: =, <>, <, >, <=, >=
  - and many other operators: see section 6.1.2 of the text and chapter 9 of the PostgreSQL documentation.
- Note that “not equals” is unusual: <>
- We can combine boolean expressions with:
  - Boolean operators: AND, OR, NOT.
Example: Compound condition

• Find 3rd- and 4th-year CSC courses:

```
SELECT * 
FROM Offering 
WHERE dept = 'CSC' AND cnum >= 300;
```
ORDER BY

• To put the tuples in order, add this as the final clause:
  ORDER BY «attribute list» [DESC]
• The default is ascending order; DESC overrides it to force descending order.
• The attribute list can include expressions: e.g.,
  ORDER BY sales+rentals
• The ordering is the last thing done before the SELECT, so all attributes are still available.
Case-sensitivity and whitespace

• Example query:

```sql
SELECT surName
FROM Student
WHERE campus = 'StG';
```

• Keywords, like `SELECT`, are not case-sensitive.
  • One convention is to use uppercase for keywords.

• Identifiers, like `Student` are not case-sensitive either.
  • One convention is to use lowercase for attributes, and a leading capital letter followed by lowercase for relations.

• Literal strings, like `'StG'`, are case-sensitive, and require single quotes.

• Whitespace (other than inside quotes) is ignored.
Expressions in SELECT clauses

• Instead of a simple attribute name, you can use an expression in a SELECT clause.
• Operands: attributes, constants
  Operators: arithmetic ops, string ops
• Examples:
  SELECT sid, grade+10 AS adjusted FROM Took;
  SELECT dept||cnum FROM course;
Expressions that are a constant

• Sometimes it makes sense for the whole expression to be a constant (something that doesn’t involve any attributes!).

• Example:

```sql
SELECT dept, cNum,
   'satisfies' AS breadthRequirement
FROM Course
WHERE breadth;
```
Pattern operators

• Two ways to compare a string to a pattern by:
  • «attribute» LIKE «pattern»
  • «attribute» NOT LIKE «pattern»

• Pattern is a quoted string
  • % means: any string
  • _ means: any single character

• Example:
  SELECT *
  FROM Course
  WHERE name LIKE ‘%Comp%’;
Aggregation
Computing on a column

• We often want to compute something across the values in a column.

• `SUM`, `AVG`, `COUNT`, `MIN`, and `MAX` can be applied to a column in a SELECT clause.

• Also, `COUNT(*)` counts the number of tuples.

• We call this aggregation.

• Note: To stop duplicates from contributing to the aggregation, use `DISTINCT` inside the brackets. (Does not affect `MIN` or `MAX`.)
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 gives us a single value per 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

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

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

- (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>
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\}
- **Example**: Tables with duplicates
Union, Intersection, and Difference

- These are expressed as:
  
  \[(«subquery») \text{ UNION } («subquery»)\]
  
  \[(«subquery») \text{ INTERSECT } («subquery»)\]
  
  \[(«subquery») \text{ EXCEPT } («subquery»)\]

- 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);
Operations $\cup$, $\cap$, and $-$ with Bags

• For $\cup$, $\cap$, and $-$ the number of occurrences of a tuple in the result requires some thought.
• (But it makes total sense.)
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` ...

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

```sql
(SELECT sid
  FROM Took
 WHERE grade > 95)
 UNION ALL
(SELECT sid
  FROM Took
 WHERE grade < 50);
```

• **Examples**: controlling-dups.txt, except-all.txt
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.
Example: defining a virtual view

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

```
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.
Outer Joins
The joins you know from RA

These can go in a FROM clause:

<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_{\text{condition}} S</td>
</tr>
</tbody>
</table>


In practice, natural join is brittle

- A working query can be broken by adding a column to a schema.
  - Example:
    
    ```
    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 practice: Don’t use natural join.
Students(sID, surName, campus)
Courses(cID, cName, WR)
Offerings(oID, cID, term, instructor, campus)
Took(sID,oID, grade)

SELECT sID, instructor
FROM Student NATURAL JOIN Took
    NATURAL JOIN Offering;
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 NULL in the other relation.

• A join that doesn’t pad with 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

<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 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>
</tbody>
</table>
Example

### R

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

### S

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

**R NATURAL LEFT 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>
</tbody>
</table>
Example

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

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<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>
Summary of join expressions

Cartesian product

A CROSS JOIN B  
\[ \text{same as } A, B \]

Theta-join

A JOIN B ON C  
✓A \{LEFT|RIGHT|FULL\} JOIN B ON C

Natural join

A NATURAL JOIN B  
✓A NATURAL \{LEFT|RIGHT|FULL\} JOIN B

✓ indicates that tuples are padded when needed.
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 is inapplicable 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.

• Implications?

• 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.
Checking for null values

• You can compare an attribute value to NULL with
  • IS NULL
  • IS NOT NULL

• Example:
  
  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.
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
The three-valued truth table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A <strong>and</strong> B</th>
<th>A <strong>or</strong> B</th>
<th>A <strong>not</strong> A</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>TF or FT</td>
<td>F</td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>TU or UT</td>
<td>U</td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FU or UF</td>
<td>F</td>
<td>U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>
Thinking of the truth-values as numbers

<table>
<thead>
<tr>
<th></th>
<th></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>1</td>
<td></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>0, 0.5</td>
<td>U</td>
<td>0.5</td>
<td>T</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>FU or UF</td>
<td>0, 0.5</td>
<td>F</td>
<td>0</td>
<td>U</td>
<td>0.5</td>
<td></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>
Thinking of the truth-values as numbers

<table>
<thead>
<tr>
<th>A</th>
<th>as a num, x</th>
<th>not A</th>
<th>1 - x</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>1</td>
<td>F</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>0</td>
<td>T</td>
<td>1</td>
</tr>
<tr>
<td>U</td>
<td>0.5</td>
<td>U</td>
<td>0.5</td>
</tr>
</tbody>
</table>
Surprises from 3-valued logic

• Some laws you are used to still hold in three-valued logic. For example,
  • **AND** is commutative.

• But others don’t. For example,
  • The law of the excluded middle breaks:
    \[(p \lor (\neg p))\] might not be **TRUE**!
  • \((0 \times x)\) might not be **0**.
Impact of null values on WHERE

• A tuple is in a query result iff the WHERE clause is **TRUE**.
• **UNKNOWN** is not good enough.
• “WHERE is picky.”
• Example: *where-null*
Impact of null values on aggregation

• Summary: Aggregation ignores \texttt{NULL}.
  • \texttt{NULL} never contributes to a sum, average, or count, and
  • Can never be the minimum or maximum of a column (unless every value is \texttt{NULL}).

• If there are no non-\texttt{NULL} values in a column, then the result of the aggregation is \texttt{NULL}.
  • Exception: \texttt{COUNT} of an empty set is 0.
# 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>min(A)</td>
<td></td>
<td>null</td>
</tr>
<tr>
<td>max(A)</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>sum(A)</td>
<td></td>
<td>all tuples count</td>
</tr>
<tr>
<td>avg(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>count(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>count(*)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
More re the impact of null values

• Other corner cases to think about:
  • `SELECT DISTINCT`: are 2 `NULL` values equal?
  • natural join: are 2 `NULL` values equal?
  • set operations: are 2 `NULL` values equal?

• And later, when we learn about constraints:
  • `UNIQUE` constraint: do 2 `NULL` values violate?

• This behaviour may vary across DBMSs.
Summary re NULL

• Any comparison with NULL yields UNKNOWN.
• WHERE is picky: it only accepts TRUE.
• Therefore NATURAL JOIN is picky too.
• Aggregation ignores NULL.
• In other situations where NULLs matter
  • when a truth-value may be NULL
  • when it matters whether two NULL are considered the same

Don’t assume. Behaviour may vary by DBMS.
Subqueries
Where can a subquery go?

- Relational algebra syntax is so elegant that it’s easy to see where subqueries can go.
- In SQL, a bit more thought is required . . .
Subqueries in a FROM clause

- 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.
Worksheet, Q1:

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.
Worksheet, Q2:

```sql
SELECT sid, surname
FROM Student
WHERE cgpa >
    (SELECT cgpa
     FROM Student
     WHERE sid = 99999);
```

- We can’t do the analogous thing in RA:

```
\Pi_{\text{sid, surname}} \sigma_{\text{cgpa} > (\text{subquery})} \text{Student}
```
Special cases

• What if the subquery returns `NULL`?

• What if the subquery could return more than one value?
Quantifying over multiple results

• When a subquery can return multiple values, we can make comparisons using a quantifier.

• Example:

```sql
SELECT sid, surname
FROM Student
WHERE cgpa >
    (SELECT cgpa
     FROM Student
     WHERE campus = 'StG');
```

• We can require that
  • cgpa > all of them, or
  • cgpa > at least one of them.
The Operator ANY

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

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

• \( x \) 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.,
\[ \forall y \in \ «\text{subquery results}\» \ | \ x \ «\text{comparison}\» y \]

• x can be a list of attributes, but this feature is not supported by psql.

• Example: any-all
The Operator IN

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

• Semantics:
  Its value is true iff \( x \) is in the set of rows generated by the subquery.

• \( x \) can be a list of attributes, and `psql` does support this feature.
Worksheet, Q3:

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');
Worksheet, Q4:

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?
The Operator EXISTS

• Syntax:
  EXISTS («subquery»)

• Semantics:
  Its value is true iff the subquery has at least one tuple.

• Read it as “exists a row in the subquery result”
Example: EXISTS

```
SELECT surname, cgpa
FROM Student
WHERE EXISTS (  
    SELECT *  
    FROM Took  
    WHERE Student.sid = Took.sid and grade > 85  
);  
```
Worksheet, Q5:

SELECT instructor
FROM Offering Off1
WHERE NOT EXISTS (  
  SELECT *  
  FROM Offering  
  WHERE  
    oid <> Off1.oid AND  
    instructor = Off1.instructor  
);
Worksheet, Q6:

```
SELECT DISTINCT oid
FROM Took
WHERE EXISTS (  
    SELECT *
    FROM Took t, Offering o
    WHERE
        t.oid = o.oid AND
        t.oid <> Took.oid AND
        o.dept = 'CSC' AND
        took.sid = t.sid )
```
Scope

• 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

```sql
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.
Modifying a Database
Database Modifications

- Queries return a relation.
- A modification command does not; it changes the database in some way.
- Three kinds of modifications:
  - Insert a tuple or tuples.
  - Delete a tuple or tuples.
  - Update the value(s) of an existing tuple or tuples.
Two ways to insert

• We’ve already seen two ways to insert rows into an empty table:

\[
\text{INSERT INTO «table» VALUES «list of rows»;}
\]

\[
\text{INSERT INTO «table» («subquery»);} 
\]

• These can also be used to add rows to a non-empty table.
Naming attributes in INSERT

- Sometimes we want to insert tuples, but we don’t have values for all attributes.
- If we name the attributes we are providing values for, the system will use NULL or a default for the rest.
- Convenient!
Example

CREATE TABLE Invite (  
  name TEXT,  
  campus TEXT DEFAULT 'StG',  
  email TEXT,  
  age INT);

INSERT INTO Invite(name, email)  
(  SELECT firstname, email  
  FROM Student  
  WHERE cgpa > 3.4  );

Here, name and email get values from the query, campus gets the default value, and age gets NULL.
Deletion

• Delete tuples satisfying a condition:
  
  \[
  \text{DELETE FROM} \ \text{«relation»} \\
  \text{WHERE} \ \text{«condition»}; \\
  \]

• Delete all tuples:
  
  \[
  \text{DELETE FROM} \ \text{«relation»}; \\
  \]
Example 1: Delete Some Tuples

DELETE FROM Course
WHERE NOT EXISTS ( 
    SELECT *
    FROM Took JOIN Offering
    ON Took.oid = Offering.oid
    WHERE
        grade > 50 AND
        Offering.dept = Course.dept AND
        Offering.cnum = Course.cnum
);

Updates

- To change the value of certain attributes in certain tuples to given values:

```
UPDATE «relation»
SET «list of attribute assignments»
WHERE «condition on tuples»;
```
Example: update one tuple

- Updating one tuple:
  ```sql
  UPDATE Student
  SET campus = 'UTM'
  WHERE sid = 99999;
  ```

- Updating several tuples:
  ```sql
  UPDATE Took
  SET grade = 50
  WHERE grade >= 47 and grade < 50;
  ```
Updates on Views

• Generally, it is impossible to modify a virtual view, because it doesn’t exist.

• Can’t we “translate” updates on views into “equivalent” updates on base tables?
  • Not always (in fact, not often).
  • Most systems prohibit most view updates.
Example: The View

- CREATE VIEW Synergy AS
- SELECT Likes.drinker, Likes.beer, Sells.bar
- FROM Likes, Sells, Frequents
- WHERE Likes.drinker = Frequents.drinker
  AND Likes.beer = Sells.beer
  AND Sells.bar = Frequents.bar;

Natural join of Likes, Sells, and Frequents

Pick one copy of each attribute
Interpreting a View Insertion

- We cannot insert into Synergy --- it is a virtual view.
- But we could try to translate a (drinker, beer, bar) triple into three insertions of projected pairs, one for each of Likes, Sells, and Frequent.
  - Sells.price will have to be NULL.
  - There isn’t always a unique translation.
Materialized Views

- Problem: each time a base table changes, the materialized view may change.
  - Cannot afford to recompute the view with each change.

- Solution: Periodic reconstruction of the materialized view, which is otherwise “out of date.”
Example: A Data Warehouse

- Wal-Mart stores every sale at every store in a database.
- Overnight, the sales for the day are used to update a data warehouse = materialized views of the sales.
- The warehouse is used by analysts to predict trends and move goods to where they are selling best.