Lecture 4:
SQL Queries
History of SQL

• Stands for *Structured Query Language*
• Developed at IBM by Donald D. Chamberlin and Raymond F. Boyce

• Originally called SEQUEL
  – Now written SQL but still pronounced “SEQUEL”

• 1\textsuperscript{st} commercial language for Codd’s model
  – First commercial system: Oracle (v2); later came IBM products based on System R
  – Stronebraker’s Ingres used QUEL, similar to SQL
    • Eventually converted to SQL

• Standardized as ANSI (1986), ISO (1987)
SQL vs. Theory

• SQL is an instantiation of the relational theory based on RA / logic foundations,
  – Yet the syntax is close to natural English
• ... with several nontrivial differences:
  – A relation is not a tuple set, but rather a tuple list
    • Repetitions are allowed
    • Order is meaningful
  – NULL values can represent missing values
    • It is not the standard true/false logic, but rather the three-valued logic (what is the meaning of NULL>5 or NULL≤5?)
• More or less the same across DBs, yet different vendors provide different extensions
Outline

• Introduction
• Basic SQL Queries
• Aggregation and Grouping
• NULLs
• Nested SQL Queries
• Views
Basic SQL Query

```
SELECT A_1, ..., A_k
FROM R_1, ..., R_n
WHERE Condition(B_1, ..., B_m)
```

List of attributes → List of relation names

- \( A_1, \ldots, A_k, B_1, \ldots, B_m \) are all in \( R_1, \ldots, R_n \)

**Example:**

```
SELECT course
FROM Student, Enroll
WHERE Student.sid = Enroll.sid
```
Basic SQL to RA

\[
\pi_{A_1, \ldots, A_k} \sigma \text{Condition}(B_1, \ldots, B_m) (R_1 \times \ldots \times R_n)
\]

Except that RA does not produce duplicates
DISTINCT Eliminates Duplicate Tuples

Eliminate duplicate tuples

\[ \pi_{A_1, \ldots, A_k} \sigma_{Condition(B_1, \ldots, B_m)} (R_1 \times \ldots \times R_n) \]

Except that RA does not produce duplicates
• As said previously, the result of an SQL query is a *list* of tuples, not a *set* as in RA

• However, SQL does not guarantee any order, unless one is specifically requested
  – We will later see how

• Hence, it is conventional to view the result as a *bag* (set with repetitions) rather than a list
Bag Semantics

• Mathematically speaking, a bag is a pair \((A, \mu)\) where \(A\) is a set and \(\mu : A \rightarrow \mathbb{N}\) associates a \textit{multiplicity} to each element in \(A\)
  – Multiplicity zero is the same as non-membership

• Bag semantics has a specialized semantics for set operations
  – \((A, \mu) \cup (B, \lambda) = (C, \xi)\) where \(C = A \cup B\) and \(\xi = \mu + \lambda\)
  – \((A, \mu) \cap (B, \lambda) = (C, \xi)\) where \(C = A \cap B\) and \(\xi = \min(\mu, \lambda)\)
  – \((A, \mu) \setminus (B, \lambda) = (C, \xi)\) where \(C = A\) and \(\xi = \max(0, \mu - \lambda)\)
Why do you think SQL architects have chosen bag semantics? Why not just simple set semantics?
Example

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>DB</td>
</tr>
<tr>
<td>861</td>
<td>PL</td>
</tr>
<tr>
<td>753</td>
<td>PL</td>
</tr>
</tbody>
</table>

```
SELECT name
FROM Student, Enroll
WHERE Student.sid = Enroll.sid
```
### Example

#### Student

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
<td>2</td>
</tr>
</tbody>
</table>

#### Enroll

<table>
<thead>
<tr>
<th>sid</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>DB</td>
</tr>
<tr>
<td>861</td>
<td>PL</td>
</tr>
<tr>
<td>753</td>
<td>PL</td>
</tr>
</tbody>
</table>

```sql
SELECT name
FROM Student, Enroll
WHERE Student.sid = Enroll.sid
```
### Example

**Student**

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
<td>2</td>
</tr>
</tbody>
</table>

**Enroll**

<table>
<thead>
<tr>
<th>sid</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>DB</td>
</tr>
<tr>
<td>861</td>
<td>PL</td>
</tr>
<tr>
<td>753</td>
<td>PL</td>
</tr>
</tbody>
</table>

```sql
SELECT name
FROM Student, Enroll
WHERE Student.sid = Enroll.sid
```
Example

<table>
<thead>
<tr>
<th>Student</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>name</td>
<td>year</td>
</tr>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Enroll</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>course</td>
</tr>
<tr>
<td>861</td>
<td>DB</td>
</tr>
<tr>
<td>861</td>
<td>PL</td>
</tr>
<tr>
<td>753</td>
<td>PL</td>
</tr>
</tbody>
</table>

```
SELECT name
FROM Student, Enroll
WHERE Student.sid = Enroll.sid
```
Example with DISTINCT

**Student**

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
<td>2</td>
</tr>
</tbody>
</table>

**Enroll**

<table>
<thead>
<tr>
<th>sid</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>DB</td>
</tr>
<tr>
<td>861</td>
<td>PL</td>
</tr>
<tr>
<td>753</td>
<td>PL</td>
</tr>
</tbody>
</table>

SELECT DISTINCT name
FROM Student, Enroll
WHERE Student.sid = Enroll.sid
More SELECT Options

• SQL allows for several important operations in the SELECT clause
  – Shorthand for selecting all attributes (*)
  – Attributes can be renamed
  – Attributes can be invented as functions of other attributes

• (Later: aggregate functions)
Example: Select All Attributes

**Student**

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
<td>2</td>
</tr>
</tbody>
</table>

**Enroll**

<table>
<thead>
<tr>
<th>sid</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>DB</td>
</tr>
<tr>
<td>861</td>
<td>PL</td>
</tr>
<tr>
<td>753</td>
<td>PL</td>
</tr>
</tbody>
</table>

```
SELECT *
FROM Student, Enroll
WHERE Student.sid = Enroll.sid
```
Example: Attribute Naming (Aliasing)

**Student**

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
<td>2</td>
</tr>
</tbody>
</table>

**Enroll**

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>DB</td>
</tr>
<tr>
<td>861</td>
<td>PL</td>
</tr>
<tr>
<td>753</td>
<td>PL</td>
</tr>
</tbody>
</table>

```sql
SELECT name AS student, cid AS course
FROM Student, Enroll
WHERE Student.sid = Enroll.sid
```
Example: Functions on Attributes

```
SELECT sid, course, credit*grade AS cg, 'great' as comment
FROM Took
WHERE grade > 69

sid  course  credit  grade
861  DB    3      80
753  PL    2      91
955  PL    2      65
```
Basic Query (select-project-join):

\[ \pi_{A_1,\ldots,A_k} \sigma_{\text{Condition}(B_1,\ldots,B_m)}(R_1 \times \ldots \times R_n) \]

SQL deploys a generalized model:

\[ \text{SELECT } F_1(t),\ldots,F_k(t) \]
\[ \text{FROM } R_1,\ldots,R_n \]
\[ \text{WHERE } \text{Condition}(B_1,\ldots,B_m) \]

where \( t \) is a tuple in \( \sigma_{\text{Condition}(B_1,\ldots,B_m)}(R_1 \times \ldots \times R_n) \)
With what we have so far, could you find the students who are enrolled in both DB and PL?
### Relation Naming (Aliasing)

**Student**

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
<td>2</td>
</tr>
</tbody>
</table>

**Enroll**

<table>
<thead>
<tr>
<th>sid</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>DB</td>
</tr>
<tr>
<td>861</td>
<td>PL</td>
</tr>
<tr>
<td>753</td>
<td>PL</td>
</tr>
</tbody>
</table>

```sql
SELECT Student.sid, name
FROM Student, Enroll E, Enroll F
WHERE Student.sid = E.sid AND
      Student.sid = F.sid AND
      E.course='DB' AND
      F.course='PL'
```

**DISTINCT makes a difference?**
SELECT Student.sid, name
FROM Student, Enroll E, Enroll F
WHERE Student.sid = E.sid AND
    Student.sid = F.sid AND
    E.course != F.course

DISTINCT makes a difference?
The WHERE Clause

• The WHERE clause allows to build arbitrary propositional logic over built-in predicates over attributes
  – Logical operators: AND, OR, NOT

• Several built-in predicates; for example:
  – Comparisons on numbers/strings (lexicographic)
    \[ =, \neq, >, <, \geq, \leq, \text{between}(x \text{ AND } y) \]
  – Membership in lists:
    \[ \text{IN}(x_1,\ldots,x_k), \quad \text{NOT} \text{IN}(x_1,\ldots,x_k) \]

• (Later: EXISTS, > ANY, > ALL, IS NULL, ...)

24
Example 1

```
SELECT *
FROM Took
WHERE grade between(70 AND 95) AND course < 'PL'
```

<table>
<thead>
<tr>
<th>sid</th>
<th>course</th>
<th>credit</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>DB</td>
<td>3</td>
<td>80</td>
</tr>
<tr>
<td>753</td>
<td>PL</td>
<td>2</td>
<td>91</td>
</tr>
<tr>
<td>955</td>
<td>PL</td>
<td>2</td>
<td>65</td>
</tr>
</tbody>
</table>

```
sid        course | credit | grade
----------|--------|-------
861        DB      | 3      | 80    
```
Example 2

**Took**

<table>
<thead>
<tr>
<th>sid</th>
<th>course</th>
<th>credit</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>DB</td>
<td>3</td>
<td>80</td>
</tr>
<tr>
<td>753</td>
<td>PL</td>
<td>2</td>
<td>91</td>
</tr>
<tr>
<td>955</td>
<td>PL</td>
<td>2</td>
<td>65</td>
</tr>
</tbody>
</table>

```
SELECT *
FROM Took
WHERE course IN ('PL', 'OS', 'AI')
```
Question

With what we have so far, could you find the ids of all persons (students and employees)?

**Student**

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
<td>2</td>
</tr>
</tbody>
</table>

**Employee**

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>233</td>
<td>Alma</td>
</tr>
<tr>
<td>651</td>
<td>Avia</td>
</tr>
<tr>
<td>122</td>
<td>Avi</td>
</tr>
</tbody>
</table>
Set Operations

• We can apply union, intersection and difference to two (or more) queries
  – \((Q_1)\) UNION (\(Q_2\))
  – \((Q_1)\) INTERSECT (\(Q_2\))
  – \((Q_1)\) EXCEPT (\(Q_2\))

• Subqueries must be union compatible in a weak sense
  – Same #attributes
  – Types of corresponding attributes must be convertible to each other (e.g., int → float)
  – The output adopts names of 1st subquery
Bag or Set Semantics?

- Default is *set semantics*:
  1. Eliminate duplicates
  2. Apply operator
  3. Eliminate duplicates

- For bag semantics, use the keyword ALL
  - \((Q_1) \text{ UNION ALL } (Q_2)\)
  - \((Q_1) \text{ INTERSECT ALL } (Q_2)\)
  - \((Q_1) \text{ EXCEPT ALL } (Q_2)\)
### Question Revisited

**Student**

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
<td>2</td>
</tr>
</tbody>
</table>

**Employee**

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>233</td>
<td>Alma</td>
</tr>
<tr>
<td>651</td>
<td>Avia</td>
</tr>
<tr>
<td>122</td>
<td>Avi</td>
</tr>
</tbody>
</table>

(SELECT sid FROM Student)

UNION

(SELECT id FROM Employee)

---

sid

<table>
<thead>
<tr>
<th>861</th>
</tr>
</thead>
<tbody>
<tr>
<td>753</td>
</tr>
<tr>
<td>955</td>
</tr>
<tr>
<td>233</td>
</tr>
<tr>
<td>651</td>
</tr>
<tr>
<td>122</td>
</tr>
</tbody>
</table>
What are the Results?

**Student**

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
<td>2</td>
</tr>
</tbody>
</table>

**Employee**

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>233</td>
<td>Alma</td>
</tr>
<tr>
<td>651</td>
<td>Avia</td>
</tr>
<tr>
<td>122</td>
<td>Avi</td>
</tr>
</tbody>
</table>

- \((\text{SELECT name FROM Student})\)
  - UNION
  - \((\text{SELECT name FROM Employee})\)

- \((\text{SELECT name FROM Student})\)
  - UNION ALL
  - \((\text{SELECT name FROM Employee})\)

- \((\text{SELECT name FROM Student})\)
  - UNION ALL
  - \((\text{SELECT name FROM Employee})\)

EXCEPT ALL

- \((\text{SELECT name FROM Employee})\)
Tuple Order

• Recall that the result of an SQL query is a list of tuples
  – But we usually ignore this order since there is no guarantee on any specific order

• You can specify an order by sort keys, and then this order is guaranteed
  – But no guarantees on ties

• And once we can control the order, we can ask for the top-k in the order
  – Simple: stop after k
SELECT $A_1,\ldots,A_k$
FROM $R_1,\ldots,R_n$
WHERE $Condition(B_1,\ldots,B_m)$
ORDER BY $C_1,\ldots,C_k$

$c_1,\ldots,c_k$ are all in $R_1,\ldots,R_n$

Example:

SELECT *
FROM Student, Enroll
WHERE Student.sid = Enroll.sid
ORDER BY course, name

<table>
<thead>
<tr>
<th>Student</th>
<th>Enroll</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>course</td>
</tr>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td></td>
</tr>
</tbody>
</table>
### Example 1

**Student**

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
<td>2</td>
</tr>
</tbody>
</table>

**Enroll**

<table>
<thead>
<tr>
<th>sid</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>DB</td>
</tr>
<tr>
<td>861</td>
<td>PL</td>
</tr>
<tr>
<td>753</td>
<td>PL</td>
</tr>
<tr>
<td>753</td>
<td>AI</td>
</tr>
<tr>
<td>753</td>
<td>DC</td>
</tr>
</tbody>
</table>

```
SELECT *
FROM Student, Enroll
WHERE Student.sid = Enroll.sid
ORDER BY name, course
```
Example 2

**Student**

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
<td>2</td>
</tr>
</tbody>
</table>

**Enroll**

<table>
<thead>
<tr>
<th>sid</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>DB</td>
</tr>
<tr>
<td>861</td>
<td>PL</td>
</tr>
<tr>
<td>753</td>
<td>PL</td>
</tr>
<tr>
<td>753</td>
<td>AI</td>
</tr>
<tr>
<td>753</td>
<td>DC</td>
</tr>
</tbody>
</table>

```
SELECT *
FROM Student, Enroll
WHERE Student.sid = Enroll.sid
ORDER BY name ASC, course DESC
```
Example 3

**Student**

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
<td>2</td>
</tr>
</tbody>
</table>

**Enroll**

<table>
<thead>
<tr>
<th>sid</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>DB</td>
</tr>
<tr>
<td>861</td>
<td>PL</td>
</tr>
<tr>
<td>861</td>
<td>PL</td>
</tr>
<tr>
<td>753</td>
<td>PL</td>
</tr>
<tr>
<td>753</td>
<td>Al</td>
</tr>
<tr>
<td>753</td>
<td>DC</td>
</tr>
<tr>
<td>753</td>
<td>PL</td>
</tr>
</tbody>
</table>

**Query**

```
SELECT Student.sid, course
FROM Student, Enroll
WHERE Student.sid = Enroll.sid
ORDER BY name, course
```
Top-k Tuples

SQL allows to limit the result to only the first $k$ answers, for some number $k$ of choice

```
SELECT $A_1,\ldots,A_k$
    FROM $R_1,\ldots,R_n$
    WHERE $Condition(B_1,\ldots,B_m)$
    ORDER BY $C_1,\ldots,C_k$
LIMIT $k$
```

Example:

```
SELECT *
    FROM Student, Enroll
    WHERE Student.sid = Enroll.sid
ORDER BY name
LIMIT 8
```
### Example

#### Student

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
<td>2</td>
</tr>
</tbody>
</table>

#### Enroll

<table>
<thead>
<tr>
<th>sid</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>DB</td>
</tr>
<tr>
<td>861</td>
<td>PL</td>
</tr>
<tr>
<td>753</td>
<td>PL</td>
</tr>
<tr>
<td>753</td>
<td>AI</td>
</tr>
<tr>
<td>753</td>
<td>DC</td>
</tr>
</tbody>
</table>

```sql
SELECT *
FROM Student, Enroll
WHERE Student.sid = Enroll.sid
ORDER BY name, course
LIMIT 3
```
Outline

• Introduction
• Basic SQL Queries
• Aggregation and Grouping
• NULLs
• Nested SQL Queries
• Views
What is the average \#likes per posting in each faculty? Show only for faculties with >2 liked postings.
Scalar vs. Aggregate Functions

• **Scalar** function: sequence-of-values to value
  – ROUND(v,i): round v to i decimals
  – UPPER(v): convert string to uppercase
  – -v, v+w, v*w, ...: arithmetic
  – NOW(): current time

• **Aggregate** function: column to value
  – SUM(C) – sum over all numbers in C
  – COUNT(C) - number of rows in C
  – AVG(C) – SUM(C)/COUNT(C)
  – MAX(C) – largest value
  – MIN(C) – smallest value
Aggregate Query

Aggregate function over a column

SELECT $\text{Agg}_1(C_1), \ldots, \text{Agg}_k(C_k)$
FROM $R_1, \ldots, R_n$
WHERE $\text{Condition}(B_1, \ldots, B_m)$

Example:

SELECT SUM(credit)
FROM Took
WHERE sid=861
Semantics of Aggregate Queries

\[
\begin{align*}
\text{SELECT } & \text{Agg}_1(c_1), \ldots, \text{Agg}_k(c_k) \\
& \text{FROM } R_1, \ldots, R_n \\
& \text{WHERE } \text{Condition}(B_1, \ldots, B_m)
\end{align*}
\]

where each \( c_i \) is a column obtained from

\[
\sigma_{\text{Condition}(B_1, \ldots, B_m)}(R_1 \times \ldots \times R_n)
\]
What counts as a column?

- Attribute name from $R_1, \ldots, R_n$
  - course, grade, ...

- Scalar over columns (row-by-row)
  - grade/2, grade*credit, UPPER(name)

- Duplicate elimination: DISTINCT C
  - DISTINCT UPPER(name)
### Example

**SELECT COUNT(sid) as num, MAX(grade) as max FROM Took WHERE course='PL'**

<table>
<thead>
<tr>
<th>sid</th>
<th>course</th>
<th>credit</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>DB</td>
<td>3</td>
<td>95</td>
</tr>
<tr>
<td>861</td>
<td>PL</td>
<td>2</td>
<td>72</td>
</tr>
<tr>
<td>753</td>
<td>PL</td>
<td>2</td>
<td>91</td>
</tr>
<tr>
<td>955</td>
<td>PL</td>
<td>2</td>
<td>65</td>
</tr>
</tbody>
</table>
## How Many Courses?

### Took

<table>
<thead>
<tr>
<th>sid</th>
<th>course</th>
<th>credit</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>DB</td>
<td>3</td>
<td>95</td>
</tr>
<tr>
<td>742</td>
<td>db</td>
<td>3</td>
<td>58</td>
</tr>
<tr>
<td>861</td>
<td>PL</td>
<td>2</td>
<td>72</td>
</tr>
<tr>
<td>753</td>
<td>PL</td>
<td>2</td>
<td>91</td>
</tr>
<tr>
<td>955</td>
<td>PL</td>
<td>2</td>
<td>65</td>
</tr>
</tbody>
</table>

1. **SELECT COUNT(course) as num**
   - **FROM Took**
   - **num** = 5

2. **SELECT COUNT(distinct course) as num**
   - **FROM Took**
   - **num** = 3

3. **SELECT COUNT(distinct UPPER(course)) as num**
   - **FROM Took**
   - **num** = 2
What Does This Compute?

```
SELECT exp(sum(ln(prob)))
FROM UnbalancedDice
```

<table>
<thead>
<tr>
<th>dice</th>
<th>prob</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.5</td>
</tr>
<tr>
<td>b</td>
<td>0.3</td>
</tr>
<tr>
<td>c</td>
<td>0.6</td>
</tr>
<tr>
<td>d</td>
<td>0.8</td>
</tr>
</tbody>
</table>
### Average Per Course

**Took**

<table>
<thead>
<tr>
<th>sid</th>
<th>course</th>
<th>credit</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>522</td>
<td>DB</td>
<td>3</td>
<td>70</td>
</tr>
<tr>
<td>753</td>
<td>PL</td>
<td>2</td>
<td>91</td>
</tr>
<tr>
<td>861</td>
<td>DB</td>
<td>3</td>
<td>80</td>
</tr>
<tr>
<td>861</td>
<td>PL</td>
<td>2</td>
<td>72</td>
</tr>
<tr>
<td>861</td>
<td>AI</td>
<td>3</td>
<td>87</td>
</tr>
<tr>
<td>955</td>
<td>PL</td>
<td>2</td>
<td>65</td>
</tr>
<tr>
<td>955</td>
<td>AI</td>
<td>3</td>
<td>96</td>
</tr>
</tbody>
</table>

**SQL Query**

- **SELECT 'DB' AS course, AVG(grade) as avg FROM Took WHERE course='DB';**
  - **course** | **avg**
    - DB | 75

- **SELECT 'PL' AS course, AVG(grade) as avg FROM Took WHERE course='PL';**
  - **course** | **avg**
    - PL | 76

- **SELECT 'AI' AS course, AVG(grade) as avg FROM Took WHERE course='AI';**
  - **course** | **avg**
    - AI | 91.5

**Problem with this solution?**
### Grouping Idea

#### SELECT \( \text{agg}(B) \)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>( v_1 )</td>
</tr>
<tr>
<td>a</td>
<td>( v_2 )</td>
</tr>
<tr>
<td>b</td>
<td>( v_3 )</td>
</tr>
<tr>
<td>b</td>
<td>( v_4 )</td>
</tr>
<tr>
<td>b</td>
<td>( v_5 )</td>
</tr>
<tr>
<td>c</td>
<td>( v_6 )</td>
</tr>
<tr>
<td>c</td>
<td>( v_7 )</td>
</tr>
</tbody>
</table>

#### \( \text{agg}( \) )

- \( v_1 \)
- \( v_2 \)
- \( v_3 \)
- \( v_4 \)
- \( v_5 \)
- \( v_6 \)
- \( v_7 \)

#### SELECT \( \text{agg}(B) \) GROUP BY A

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>( v_1 )</td>
</tr>
<tr>
<td>a</td>
<td>( v_2 )</td>
</tr>
<tr>
<td>b</td>
<td>( v_3 )</td>
</tr>
<tr>
<td>b</td>
<td>( v_4 )</td>
</tr>
<tr>
<td>b</td>
<td>( v_5 )</td>
</tr>
<tr>
<td>c</td>
<td>( v_6 )</td>
</tr>
<tr>
<td>c</td>
<td>( v_7 )</td>
</tr>
</tbody>
</table>

#### \( \text{agg}( \) )

- \( v_1 \)
- \( v_2 \)
- \( v_3 \)
- \( v_4 \)
- \( v_5 \)
- \( v_6 \)
- \( v_7 \)
### Group Selection

#### SELECT `agg(B)`
#### GROUP BY A
#### HAVING A > a

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>v1</td>
</tr>
<tr>
<td>a</td>
<td>v2</td>
</tr>
<tr>
<td>b</td>
<td>v3</td>
</tr>
<tr>
<td>b</td>
<td>v4</td>
</tr>
<tr>
<td>b</td>
<td>v5</td>
</tr>
<tr>
<td>c</td>
<td>v6</td>
</tr>
<tr>
<td>c</td>
<td>v7</td>
</tr>
</tbody>
</table>

#### SELECT `agg(B)`
#### GROUP BY A

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>v1</td>
</tr>
<tr>
<td>a</td>
<td>v2</td>
</tr>
<tr>
<td>b</td>
<td>v3</td>
</tr>
<tr>
<td>b</td>
<td>v4</td>
</tr>
<tr>
<td>b</td>
<td>v5</td>
</tr>
<tr>
<td>c</td>
<td>v6</td>
</tr>
<tr>
<td>c</td>
<td>v7</td>
</tr>
</tbody>
</table>

50
**Grouping Syntax**

```
SELECT Columns(G₁,...,Gₗ,...,agg₁,...,aggₚ)
FROM R₁,...,Rₙ
WHERE Condition(B₁,...,Bₘ)
GROUP BY G₁,...,Gₗ
HAVING Condition(G₁,...,Gₗ,...,agg₁,...,aggₚ)
```

Columns obtained only from grouping attributes and aggregate functions

**Why do we need both WHERE and HAVING?**

Sometimes…

Attention testing
### Example

**Student**

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
<td>2</td>
</tr>
<tr>
<td>699</td>
<td>Adi</td>
<td>3</td>
</tr>
<tr>
<td>729</td>
<td>Avia</td>
<td>3</td>
</tr>
</tbody>
</table>

**Took**

<table>
<thead>
<tr>
<th>sid</th>
<th>course</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>DB</td>
<td>79</td>
</tr>
<tr>
<td>861</td>
<td>PL</td>
<td>82</td>
</tr>
<tr>
<td>753</td>
<td>PL</td>
<td>93</td>
</tr>
<tr>
<td>955</td>
<td>AI</td>
<td>72</td>
</tr>
<tr>
<td>955</td>
<td>DB</td>
<td>99</td>
</tr>
</tbody>
</table>

**Course**

<table>
<thead>
<tr>
<th>name</th>
<th>credit</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB</td>
<td>3</td>
</tr>
<tr>
<td>PL</td>
<td>2</td>
</tr>
<tr>
<td>AI</td>
<td>3</td>
</tr>
</tbody>
</table>

**Task:** Find the average grade of 2\textsuperscript{nd} year students with at least 5 credit points.

\[
\text{Average grade} = \frac{(79 \times 3 + 82 \times 2)}{5}
\]

<table>
<thead>
<tr>
<th>name</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alma</td>
<td>80.2</td>
</tr>
</tbody>
</table>

... ...
Aggregate Functions

Task: Find the average grade of 2\textsuperscript{nd} year students with at least 5 credit points.

```
SELECT ...  
FROM Student S, Took T, Course C  
WHERE S.sid=T.sid AND T.course = C.name
```
### Aggregate Functions

#### Task
Find the average grade of 2\textsuperscript{nd} year students with at least 5 credit points.

```sql
SELECT ...
FROM Student S, Took T, Course C
WHERE S.sid=T.sid AND T.course = C.name
GROUP BY S.sid
```
Task: Find the average grade of 2nd year students with at least 5 credit points.

```
SELECT ... FROM Student S, Took T, Course C
WHERE S.sid=T.sid AND T.course = C.name
GROUP BY S.sid
HAVING S.year>=2 AND SUM(C.credit)>=5;
```
Aggregate Functions

Task: Find the average grade of 2nd year students with at least 5 credit points.

```
SELECT ...
FROM Student S, Took T, Course C
WHERE S.sid=T.sid AND T.course = C.name
GROUP BY S.sid, S.year
HAVING S.year>=2 AND SUM(C.credit)>=5;
```
Task: Find the average grade of 2nd year students with at least 5 credit points.

```
SELECT S.name, 
    sum(T.grade*C.credit)*1.0/sum(C.credit) as average 
FROM Student S, Took T, Course C 
WHERE S.sid=T.sid AND T.course = C.name 
GROUP BY S.sid, S.year 
HAVING S.year>=2 AND SUM(C.credit)>=5;
```
Aggregate Functions

<table>
<thead>
<tr>
<th>Student</th>
<th>Took</th>
<th>Course</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>sid</td>
<td>name</td>
</tr>
<tr>
<td>name</td>
<td>course</td>
<td>credit</td>
</tr>
<tr>
<td>year</td>
<td>grade</td>
<td></td>
</tr>
</tbody>
</table>

Task: Find the average grade of 2\textsuperscript{nd} year students with at least 5 credit points.

```sql
SELECT S.name, 
    sum(T.grade*C.credit)*1.0/sum(C.credit) as average
FROM Student S, Took T, Course C
WHERE S.sid=T.sid AND T.course = C.name
GROUP BY S.sid, S.year, S.name
HAVING S.year=2 AND SUM(C.credit)>=5;
```
What is the average #likes per posting in each faculty? Show only for faculties with >2 liked postings.
Outline

- Introduction
- Basic SQL Queries
- Aggregation and Grouping
- NULLs
- Nested SQL Queries
- Views
• Problem: pieces of data missing, but we need to keep whatever partial knowledge we have

<table>
<thead>
<tr>
<th>Enroll</th>
<th>Course</th>
</tr>
</thead>
<tbody>
<tr>
<td>student</td>
<td>course</td>
</tr>
<tr>
<td>Ahuva</td>
<td>PL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>course</th>
<th>lecturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL</td>
<td>Eran</td>
</tr>
</tbody>
</table>

• A source tells us that Alon is a student of Keren
  – How can we represent it in our database?

<table>
<thead>
<tr>
<th>Enroll</th>
<th>Course</th>
</tr>
</thead>
<tbody>
<tr>
<td>student</td>
<td>course</td>
</tr>
<tr>
<td>Ahuva</td>
<td>PL</td>
</tr>
<tr>
<td>Alon</td>
<td>⊥</td>
</tr>
</tbody>
</table>

⊥=NULL
NULL is SQL’s special “missing value”

Same queries as complete tables, but SQL assigns a special behavior to logic over NULL

– “Three-valued logic”: true, false, unknown

Alas, there are some issues...
Try It Yourself (PostgreSQL)

CREATE TABLE Enroll(
    student varchar(40),
    course varchar(40));

INSERT INTO Enroll VALUES ('Ahuva','PL'),
('Alon',NULL);

CREATE TABLE Course(
    course varchar(40),
    lecturer varchar(40));

INSERT INTO Course VALUES
('PL','Eran'), (NULL,'Keren');

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ahuva</td>
<td>PL</td>
</tr>
<tr>
<td>Alon</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>course</th>
<th>lecturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL</td>
<td>Eran</td>
</tr>
<tr>
<td></td>
<td>Keren</td>
</tr>
</tbody>
</table>

SELECT student, lecturer
FROM Enroll R, Course C
WHERE R.course = C.course;

Of course, we've lost our initial association (join)...

<table>
<thead>
<tr>
<th>student</th>
<th>lecturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ahuva</td>
<td>Eran</td>
</tr>
</tbody>
</table>
Try More Yourself (psql)

### Enroll

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ahuva</td>
<td>PL</td>
</tr>
<tr>
<td>Alon</td>
<td></td>
</tr>
</tbody>
</table>

### Course

<table>
<thead>
<tr>
<th>course</th>
<th>lecturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL</td>
<td>Eran</td>
</tr>
<tr>
<td></td>
<td>Keren</td>
</tr>
</tbody>
</table>

```
SELECT student FROM Enroll;

student
<table>
<thead>
<tr>
<th>Ahuva</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alon</td>
</tr>
</tbody>
</table>
```

```
SELECT student FROM Enroll WHERE course='PL';

student
| Ahuva |
```

```
SELECT student FROM Enroll WHERE course!='PL';

student
| Ahuva |
```

```
(SELECT student FROM Enroll WHERE course='PL')
UNION
(SELECT student FROM Enroll WHERE course!='PL');

student
| Ahuva |
```

```
SELECT student FROM Enroll WHERE course='PL' OR course!='PL';

student
| Ahuva |
```

Alon??
```
Conditions with NULL

• Principle: atomic predicates (e.g., comparison between two numbers) result in unknown (denoted U) when one or more operands is NULL
  – 5>NULL, NULL=NULL, etc.

• Then, propositional logic over atomic predicates follows the three-valued logic (3VL):
Three-Valued Logic (3VL)

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x ∧ y</th>
<th>x ∨ y</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>U</td>
<td>U</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>U</td>
<td>T</td>
<td>U</td>
<td>T</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>F</td>
<td>F</td>
<td>U</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>U</td>
<td>F</td>
<td>U</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>¬x</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

66
What problem of 3VL does this example show?

Any suggestion for an alternative semantics?

(Attend advanced course)
**Avoiding Nulls**

```sql
CREATE TABLE Enroll(
    sid int,
    course text NOT NULL
)
```

**DDL: constrain on non-nullity**

```sql
SELECT sid
FROM Enroll
WHERE course IS NOT NULL
```

```sql
(SELECT sid FROM Enroll)
EXCEPT
(SELECT sid FROM Enroll
WHERE course IS NULL)
```

**Queries: nullity testing**
Outer Joins

### Task: Extend student with the address information (convenience, join avoidance)

<table>
<thead>
<tr>
<th>SSN</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
</tr>
<tr>
<td>852</td>
<td>Ahuva</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SSN</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Haifa</td>
</tr>
<tr>
<td>302</td>
<td>Nazareth</td>
</tr>
<tr>
<td>955</td>
<td>Isfiya</td>
</tr>
<tr>
<td>852</td>
<td>Afek</td>
</tr>
</tbody>
</table>

We’ve lost Amir!
Left Outer Join

**Task:** Extend student with the address information (convenience, join avoidance)
Definition

Let \( r \) and \( s \) be relations over schemas \( R(A_1,\ldots,A_n) \) and \( S(B_1,\ldots,B_m) \), respectively.

We define:

\[
(\mathcal{r} \bowtie \mathcal{s}) = (r \bowland s) \cup ((r - \pi_{A_1,\ldots,A_n}(r \bowland s)) \times \{ (\bot,\ldots,\bot) \})
\]

That is, \( \mathcal{r} \bowtie \mathcal{s} \) contains:
- All the tuples in the join of \( r \) and \( s \)
- All the tuples of \( r \) that cannot be joined, padded with NULLs
### Right Outer Join

#### Student

<table>
<thead>
<tr>
<th>SSN</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
</tr>
<tr>
<td>852</td>
<td>Ahuva</td>
<td>2</td>
</tr>
</tbody>
</table>

#### Address

<table>
<thead>
<tr>
<th>SSN</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Haifa</td>
</tr>
<tr>
<td>302</td>
<td>Nazareth</td>
</tr>
<tr>
<td>955</td>
<td>Isfiya</td>
</tr>
<tr>
<td>852</td>
<td>Afek</td>
</tr>
</tbody>
</table>

\[
(r \bowtie s) = (r \bowtie s) \cup \left( \{(\bot, \ldots, \bot)\} \times (s - \pi_{B_1, \ldots, B_m} (r \bowtie s)) \right)
\]
### Full Outer Join

#### Student

<table>
<thead>
<tr>
<th>SSN</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
</tr>
<tr>
<td>852</td>
<td>Ahuva</td>
<td>2</td>
</tr>
</tbody>
</table>

#### Address

<table>
<thead>
<tr>
<th>SSN</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Haifa</td>
</tr>
<tr>
<td>302</td>
<td>Nazareth</td>
</tr>
<tr>
<td>955</td>
<td>Isfiya</td>
</tr>
<tr>
<td>852</td>
<td>Afek</td>
</tr>
</tbody>
</table>

#### Full Outer Join Calculation

$$r \bowtie s = (r \bowtie s) \cup (r \bowtie s)$$

<table>
<thead>
<tr>
<th>SSN</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
</tr>
<tr>
<td>852</td>
<td>Ahuva</td>
<td>2</td>
</tr>
</tbody>
</table>

$$r \bowtie s$$

<table>
<thead>
<tr>
<th>SSN</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Haifa</td>
</tr>
<tr>
<td>302</td>
<td>Nazareth</td>
</tr>
<tr>
<td>955</td>
<td>Isfiya</td>
</tr>
<tr>
<td>852</td>
<td>Afek</td>
</tr>
</tbody>
</table>

$$= (r \bowtie s) \cup (r \bowtie s)$$

<table>
<thead>
<tr>
<th>SSN</th>
<th>name</th>
<th>year</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
<td>Haifa</td>
</tr>
<tr>
<td>302</td>
<td>Amir</td>
<td>1</td>
<td>Nazareth</td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
<td>2</td>
<td>Afek</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
<td>Isfiya</td>
</tr>
</tbody>
</table>

73
Left Outer Join in SQL

**Student**

<table>
<thead>
<tr>
<th>SSN</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
</tr>
<tr>
<td>852</td>
<td>Ahuva</td>
<td>2</td>
</tr>
</tbody>
</table>

**Address**

<table>
<thead>
<tr>
<th>SSN</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Haifa</td>
</tr>
<tr>
<td>302</td>
<td>Nazareth</td>
</tr>
<tr>
<td>955</td>
<td>Isfiya</td>
</tr>
<tr>
<td>852</td>
<td>Afek</td>
</tr>
</tbody>
</table>

SELECT S.SSN, name, year, city
FROM Student S LEFT JOIN Address A
ON (S.SSN = A.SSN)
### Right Outer Join in SQL

**Student**

<table>
<thead>
<tr>
<th>SSN</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
</tr>
<tr>
<td>852</td>
<td>Ahuva</td>
<td>2</td>
</tr>
</tbody>
</table>

**Address**

<table>
<thead>
<tr>
<th>SSN</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Haifa</td>
</tr>
<tr>
<td>302</td>
<td>Nazareth</td>
</tr>
<tr>
<td>955</td>
<td>Isfiya</td>
</tr>
<tr>
<td>852</td>
<td>Afek</td>
</tr>
</tbody>
</table>

```sql
SELECT A.SSN, name, year, city
FROM Person S
RIGHT OUTER JOIN Address A
ON (S.SSN = A.SSN)
```

<table>
<thead>
<tr>
<th>SSN</th>
<th>name</th>
<th>year</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
<td>Haifa</td>
</tr>
<tr>
<td>852</td>
<td>Ahuva</td>
<td>2</td>
<td>Afek</td>
</tr>
<tr>
<td>302</td>
<td></td>
<td></td>
<td>Nazareth</td>
</tr>
<tr>
<td>955</td>
<td></td>
<td></td>
<td>Isfiya</td>
</tr>
</tbody>
</table>
# Full Outer Join in SQL

## Student

<table>
<thead>
<tr>
<th>SSN</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
</tr>
<tr>
<td>852</td>
<td>Ahuva</td>
<td>2</td>
</tr>
</tbody>
</table>

## Address

<table>
<thead>
<tr>
<th>SSN</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Haifa</td>
</tr>
<tr>
<td>302</td>
<td>Nazareth</td>
</tr>
<tr>
<td>955</td>
<td>Isfiya</td>
</tr>
<tr>
<td>852</td>
<td>Afek</td>
</tr>
</tbody>
</table>

```sql
SELECT *
FROM Person S FULL OUTER JOIN Address A
ON (S.SSN = A.SSN)
```
Outline

• Introduction
• Basic SQL Queries
• Aggregation and Grouping
• NULLs
• Nested SQL Queries
• Views
Nesting

- Nesting: one query is nested in another query as a relation/value component
- The nested query is called a **subquery**
- Where are we nesting?
  - **SELECT**
    - Select a value from a subquery
  - **FROM**
    - Use a subquery instead of an existing relation
  - **WHERE**
    - Conditions phrased via subqueries
Notation

- We denote a query that returns a single column by $N \times 1$
- We denote a query that returns a single column and at most one row as $1 \times 1$
- Subqueries (nested queries) are sometimes required to be $N \times 1$ or $1 \times 1$
  - In PostgreSQL, $1 \times 1$ is checked at runtime, hence, this property is sensitive to the database
    - As opposed to $N \times 1$
- To denote that there is no restriction we will use $N \times M$
Subquery in WHERE

- Most common place for subqueries

- Several forms:
  - As any scalar value (1x1)
    - T.grade >= (SELECT MAX(grade) FROM Took)
    - Empty result is treated as NULL
  - Membership testing (Nx1)
    - T.course IN (SELECT name FROM Course WHERE credit=2)
  - Nonemptiness testing (NxM)
    - EXISTS (SELECT * FROM Course WHERE credit=2)
Example

**Task:** Find the students that got the maximal grades

```
SELECT S.name
FROM Student S, Took T
WHERE S.sid=T.sid AND
T.grade >= (SELECT MAX(grade) from Took)
```

1x1
Task: Find the students that got the maximal grade *in some course*

<table>
<thead>
<tr>
<th>Student</th>
<th>Took</th>
<th>Course</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>name</td>
<td>year</td>
</tr>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
<td>2</td>
</tr>
<tr>
<td>699</td>
<td>Adi</td>
<td>3</td>
</tr>
<tr>
<td>729</td>
<td>Avia</td>
<td>3</td>
</tr>
</tbody>
</table>

Next…
• In a WHERE subquery, record names from the FROM clauses of enclosing queries are accessible as constants
  – In PL terminology, the subquery is within the scope of the super-query’s FROM variables

• *How does that help in our example?*
Example Revisited

Student
sid     name     year

Took
sid     course   grade

Course
course  credit

Task: Find the students that got the maximal grades

Previous example:

SELECT S.name  
FROM Student S, Took T  
WHERE S.sid=T.sid AND  
T.grade >= (SELECT MAX(grade) FROM Took)

Task: Find the students that got the maximal grade in some course

SELECT S.name  
FROM Student S, Took T  
WHERE S.sid=T.sid AND  
T.grade >= (SELECT MAX(grade) FROM Took  
WHERE Took.course=T.course)
Task: Find the students who attended courses with >100 students

```
SELECT S.name
FROM Student S, Took T
WHERE S.sid=T.sid AND
  T.course IN (
    SELECT course
    FROM Took
    GROUP BY course
    HAVING COUNT(*)>100
  )
```
Example with EXISTS

Task: Find the students who attended courses that Alma took

```
SELECT S.name
FROM Student S, Took T
WHERE S.sid=T.sid AND
  EXISTS (SELECT *
           FROM Student, Took
           WHERE Student.sid=Took.sid AND
           course=T.course and name='Alma')
```
**Example of NOT IN**

<table>
<thead>
<tr>
<th>Student</th>
<th>Took</th>
<th>Course</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>name</td>
<td>year</td>
</tr>
<tr>
<td>course</td>
<td>credit</td>
<td></td>
</tr>
</tbody>
</table>

**Task:** Find the students who attended courses that Alma **did not** take

```
SELECT S.name
FROM Student S, Took T
WHERE S.sid=T.sid AND
  T.course NOT IN (  
    SELECT T1.course
    FROM Student S1, Took T1
    WHERE S1.sid=T1.sid AND
      S1.name='Alma'   
  )
```

87
Task: Find the students who attended courses that Alma did not take

```
SELECT S.name
FROM Student S, Took T
WHERE S.sid=T.sid AND
    NOT EXISTS (  
        SELECT *
        FROM Student, Took
        WHERE Student.sid=Took.sid AND
        course=T.course and name='Alma')
```
• You can have 1x1 subqueries in the SELECT clause
• And as in WHERE, the subquery is in the scope of the FROM variables
Example of SELECT Nesting

Task: For each student-course, list student name, course, and max grade in the course

```
SELECT name, course, (SELECT MAX(grade) FROM Took WHERE course=T.course) AS MC
FROM Student S, Took T
WHERE S.sid=T.sid;
```
Nesting Inside FROM

- You can have NxM subqueries in the FROM clause
- Such a query acts as an ordinary input relation
- A subquery must be named (AS ...)  
  – Why?
- Adjacent FROM variables are not in the scope of the subquery!
Example of FROM Nesting

Task: For each student, find a course that she hasn’t taken, but at least 100 students have taken already

SELECT S.name, C.course
FROM Student S, (SELECT course
               FROM Took
               GROUP BY course
               HAVING COUNT(*)>100) AS C
WHERE NOT EXISTS ( SELECT * FROM Took WHERE
                  Took.sid = S.sid AND Took.course = C.course)
Outline

• Introduction
• Basic SQL Queries
• Aggregation and Grouping
• NULLs
• Nested SQL Queries
• Views
Practical Problem Related to Example

• **Problem 1**
  - Almost every interesting question we have requires joining Student and Took
  - Complicates queries
  - Joint computation is not shared
  - *Nevertheless, we do not want to maintain data in the joint form*
Another Problem

Problem 2

- Scenario: a student asks for a project; I ask her to implement course suggestion for other students
- I give her access: GRANT SELECT on Took to ahuva
- I can live with her seeing who took what, but not the grades!
Definition: A **view** is a stored query that can be accessed as an ordinary relation.

```
CREATE VIEW STC as
SELECT S.sid, T.course, C.credit
FROM Student S, Took T, Course C
WHERE S.sid=T.sid AND T.course=C.course
```

```
SELECT sid, course FROM STC WHERE ...
```

```
GRANT SELECT on STC to ahuva
```
Advantages of Views

• Always updated, always correct with respect to its definition
  – No need to update the view once source relations are updated

• Allows for simpler queries without introducing redundant dependencies

• For a complicated view, the chance of a mistake is smaller than that of repeated subqueries
View Management

• Two approaches to view management:
  – **Materialized view**: the view exists and constantly maintained by the system
  – **Non-materialized view**: the view is created as part of the query (default in Postgres)

<table>
<thead>
<tr>
<th></th>
<th>Non-materialized</th>
<th>Materialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slower queries</td>
<td></td>
<td>Faster queries</td>
</tr>
<tr>
<td>No extra update overhead</td>
<td></td>
<td>Slower updates</td>
</tr>
<tr>
<td>No extra storage overhead</td>
<td></td>
<td>Storage overhead</td>
</tr>
</tbody>
</table>

• *Incremental view maintenance* is an active and deep technological and research topic
  – Another fascinating topic: updating the database by updating the view (a.k.a. *view updates*)
Some Aggregation-related questions
Can you have a SELECT attribute not in the GroupBy columns?

NO

CS=> select a, c from checkagg group by a;

ERROR: column "checkagg.c" must appear in the GROUP BY clause or be used in an aggregate function

CS=> select c from checkagg group by a;

ERROR: column "checkagg.c" must appear in the GROUP BY clause or be used in an aggregate function
Can you have an aggregate on columns not in the GroupBy columns?

**YES**

| a | b | c |
|---+---+---|
| 1 | 2 | 3 |
| 1 | 2 | 5 |
| 1 | 2 | 7 |
| 2 | 3 | 7 |
| 2 | 3 | 70 |
| 2 | 3 | 80 |

(6 rows)

```
cs=> select max(c) from checkagg group by a;
     max
-----
    80
    7
(2 rows)
```

```
cs=> select a, max(c) from checkagg group by a;
    a | max
-----
    2 | 80
    1 | 7
(2 rows)
```
CS=> select * from checkagg;

<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>1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>70</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>80</td>
</tr>
</tbody>
</table>

(6 rows)

Can you have an aggregate on a GroupBy columns?

YES

CS=> select max(a) from checkagg group by a;

max

-----

  2
  1

(2 rows)

Can you have an aggregate involving GroupBy and other columns?

YES

CS=> select max(a+b+c) from checkagg group by a;

max

-----

  85
  10

(2 rows)
• easy play with sql http://sqlfiddle.com/
  http://www-cs-

• NULLS in SQL
  students.stanford.edu/~wlam/compsci/sqlnulls