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”
- 1st commercial language for Codd’s model
  - First commercial system: Oracle (v2); later came IBM products based on System R
  - Stonebraker’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\textsubscript{1}, ..., A\textsubscript{k})
FROM R\textsubscript{1}, ..., R\textsubscript{n}
WHERE \textit{Condition}(B\textsubscript{1}, ..., B\textsubscript{m})
```

Example:
```
SELECT course
FROM Student, Enroll
WHERE Student.sid = Enroll.sid
```

Basic SQL to RA

```
\pi_{A\textsubscript{1}, ..., A\textsubscript{k}}\sigma_{\textit{Condition}(B\textsubscript{1}, ..., B\textsubscript{m})}(R\textsubscript{1} \times \ldots \times R\textsubscript{n})
```

Except that RA does not produce duplicates
DISTINCT Eliminates Duplicate Tuples

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

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

Except that RA does not produce duplicates

Lists, Sets, Bags

- 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 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 \(\xi(m) = \mu(m) + \lambda(m)\).
  - \((A, \mu) \cap (B, \lambda) = (C, \xi)\) where \(C = A \cap B\) and \(\xi(m) = \min(\mu(m), \lambda(m))\).
  - \((A, \mu) \setminus (B, \lambda) = (C, \xi)\) where \(C = A\) and \(\xi(m) = \max(0, \mu(m) - \lambda(m))\).

Why do you think SQL architects have chosen bag semantics? Why not just simple set semantics?

Example

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

```
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 name
FROM Student, Enroll
WHERE Student.sid = Enroll.sid
```

```
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>PL</td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>Student</th>
<th>Enroll</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>name</td>
</tr>
<tr>
<td>861</td>
<td>Alma</td>
</tr>
<tr>
<td>753</td>
<td>Ahuva</td>
</tr>
</tbody>
</table>

Example

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

<table>
<thead>
<tr>
<th>Student</th>
<th>Enroll</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>name</td>
</tr>
<tr>
<td>861</td>
<td>Alma</td>
</tr>
<tr>
<td>753</td>
<td>Ahuva</td>
</tr>
</tbody>
</table>

Example with DISTINCT

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

<table>
<thead>
<tr>
<th>Student</th>
<th>Enroll</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>name</td>
</tr>
<tr>
<td>861</td>
<td>Alma</td>
</tr>
<tr>
<td>753</td>
<td>Ahuva</td>
</tr>
</tbody>
</table>

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

```sql
SELECT *
FROM Student, Enroll
WHERE Student.sid = Enroll.sid
```

Example: Attribute Naming (Aliasing)

```sql
SELECT student, course
FROM Student, Enroll
WHERE Student.sid = Enroll.sid
```

<table>
<thead>
<tr>
<th>Student</th>
<th>Enroll</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>name</td>
</tr>
<tr>
<td>861</td>
<td>Alma</td>
</tr>
<tr>
<td>753</td>
<td>Ahuva</td>
</tr>
</tbody>
</table>
Example: Functions on Attributes

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

<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>

How Come?

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

Basic Query (select-project-join):

SQL deploys a generalized model:

```
SELECT F_1(t),...,F_k(t) FROM R_1,...,R_n WHERE Condition(B_1,...,B_m)
```

π_{A_1},...,A_k σ_{Condition}(B_1,...,B_m) (R_1 × ... × R_n)

Relation Naming (Aliasing)

```
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?

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)
    - =, !=, >, >=, <, <=, between(x AND y)
  - Membership in lists:
    - IN(x_1,...,x_k), NOT IN(x_1,...,x_k)
- (Later: EXISTS, > ANY, > ALL, IS NULL, ...)

With what we have so far, could you find the students who are enrolled to both DB and PL?

```
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>
```

DISTINCT makes a difference?
Example 1

Example 2

Question

Set Operations

Bag or Set Semantics?

Question Revisited
What are the Results?

<table>
<thead>
<tr>
<th>Student</th>
<th>(SELECT name FROM Student) UNION (SELECT name FROM Employee)</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Employee</th>
<th>(SELECT name FROM Student) UNION ALL (SELECT name FROM Employee)</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 name FROM Student) UNION (SELECT name FROM Employee) UNION ALL (SELECT name FROM Employee) EXCEPT ALL (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

Example 1

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

Example 2

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

Example 3

```
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, ..., A_k
FROM R_1, ..., R_n
WHERE Condition(B_1, ..., B_m)
ORDER BY C_1, ..., C_k
LIMIT k
```

Example:

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

Example:

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

Example:

```
Student
sid | name  | year
--- |------ |---
861 | Alma  | 2
753 | Amir  | 1
955 | Ahuva | 2

Enroll
sid | course
--- |-------
861 | DB
861 | PL
753 | AI
753 | DC
```

Outline

- Introduction
- Basic SQL Queries
  - Aggregation and Grouping
  - NULLs
  - Nested SQL Queries
  - Views

Scalar vs. Aggregate Functions

- **Scalar** function: sequence-of-values to value
  - \( \text{ROUND}(v, i) \): round \( v \) to \( i \) decimals
  - \( \text{UPPER}(v) \): convert string to uppercase
  - \( -v, v+w, v*w, \ldots \): arithmetic
  - \( \text{NOW}() \): current time

- **Aggregate** function: column to value
  - \( \text{SUM}(C) \): sum over all numbers in \( C \)
  - \( \text{COUNT}(C) \): number of rows in \( C \)
  - \( \text{AVG}(C) \): \( \text{SUM}(C)/\text{COUNT}(C) \)
  - \( \text{MAX}(C) \): largest value
  - \( \text{MIN}(C) \): smallest value

Aggregate Query

```
SELECT \( \text{Agg}(C_i), \ldots, \text{Agg}(C_k) \)
FROM \( R_1, \ldots, R_n \)
WHERE Condition(B_1, ..., B_m)
```

Example:

```
SELECT SUM(credit)
FROM Took
WHERE sid=861
```

Task (for the end of this part)

```
What is the average #likes per posting in each faculty? Show only for faculties with >2 liked postings

Pairs / groups allowed. Email solution to me:
- bennyk@cs.technion.ac.il
```

Example:

```
Student
sid | name  | year | Enroll.sid | course
--- |------ |--- |---------- |-------
861 | Alma  | 2 | 861        | DB
861 | Alma  | 2 | 861        | PL
753 | Amir  | 1 | 753        | AI
753 | Amir  | 1 | 753        | DC
```

Example:

```
Student
sid | name  | Faculty
--- |------ |--------
861 | Alma  | CS
861 | Alma  | CS
753 | Amir  | EE
753 | Amir  | EE
```

Example:

```
Pos7ng
id | owner
--- |------
23 | Alma
45 | Alma
76 | Ahuva
79 | Ahuva
```

Example:

```
Likes
student | pos7ng
---------|------
Alma     | 45
Alma     | 76
Ahuva    | 23
Ahuva    | 76
```
Semantics of Aggregate Queries

\[
\text{SELECT } \text{Agg}(c_1), \ldots, \text{Agg}(c_k) \\
\text{FROM } R_1, \ldots, R_n \\
\text{WHERE Condition}(B_1, \ldots, B_m)
\]

where each \( c_i \) is a column obtained from
\[ c_{\text{Condition}(B_1, \ldots, B_m)}(R_1 \times \ldots \times R_n) \]

Columns

- 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>3</td>
<td>91</td>
</tr>
<tr>
<td>955</td>
<td>PL</td>
<td>2</td>
<td>65</td>
</tr>
</tbody>
</table>
```

task (for the end of this part)

Student name Faculty id owner
Alma CS 23 Alma
Ahuva EE 76 Ahuva
Amir CS 45 Amir
Ahuva EE 76 Ahuva

Posting

<table>
<thead>
<tr>
<th>student</th>
<th>posting</th>
<th>likes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alma</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>Ahuva</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>Ahuva</td>
<td>79</td>
<td></td>
</tr>
</tbody>
</table>

What is the average #likes per posting in each faculty? Show only for faculties with >2 liked postings

Pairs / groups allowed. Email solution to me: bennyk@cs.technion.ac.il
Order of Operations

1. Cartesian product (FROM)
2. Selection (WHERE)
3. Grouping (GROUP BY)
4. Group selection (HAVING)
5. Projection + aggregates (SELECT)

Example

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ahuva</td>
<td>DB</td>
<td>2</td>
</tr>
<tr>
<td>Alma</td>
<td>DB</td>
<td>1</td>
</tr>
<tr>
<td>Amir</td>
<td>PL</td>
<td>3</td>
</tr>
<tr>
<td>Avia</td>
<td>PL</td>
<td>2</td>
</tr>
<tr>
<td>Adi</td>
<td>AI</td>
<td>1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Course</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB</td>
<td>85.2</td>
</tr>
</tbody>
</table>

Average Per Course

<table>
<thead>
<tr>
<th>Task</th>
<th>sid</th>
<th>course</th>
<th>credit</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Took</td>
<td>522</td>
<td>DB</td>
<td>3</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>733</td>
<td>PL</td>
<td>2</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>861</td>
<td>DB</td>
<td>3</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>861</td>
<td>PL</td>
<td>2</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>861</td>
<td>AI</td>
<td>2</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>955</td>
<td>PL</td>
<td>2</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>955</td>
<td>AI</td>
<td>2</td>
<td>96</td>
</tr>
</tbody>
</table>

Group Selection

<table>
<thead>
<tr>
<th>Task</th>
<th>sid</th>
<th>course</th>
<th>credit</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Took</td>
<td>522</td>
<td>DB</td>
<td>3</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>733</td>
<td>PL</td>
<td>2</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>861</td>
<td>DB</td>
<td>3</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>861</td>
<td>PL</td>
<td>2</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>861</td>
<td>AI</td>
<td>2</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>955</td>
<td>PL</td>
<td>2</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>955</td>
<td>AI</td>
<td>2</td>
<td>96</td>
</tr>
</tbody>
</table>

Problem with this solution?

Grouping Idea

<table>
<thead>
<tr>
<th>Task</th>
<th>sid</th>
<th>course</th>
<th>credit</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Took</td>
<td>522</td>
<td>DB</td>
<td>3</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>733</td>
<td>PL</td>
<td>2</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>861</td>
<td>DB</td>
<td>3</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>861</td>
<td>PL</td>
<td>2</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>861</td>
<td>AI</td>
<td>2</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>955</td>
<td>PL</td>
<td>2</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>955</td>
<td>AI</td>
<td>2</td>
<td>96</td>
</tr>
</tbody>
</table>

Grouping Syntax

Columns obtained only from grouping attributes and aggregate functions

<table>
<thead>
<tr>
<th>Task</th>
<th>sid</th>
<th>course</th>
<th>credit</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Took</td>
<td>522</td>
<td>DB</td>
<td>3</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>733</td>
<td>PL</td>
<td>2</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>861</td>
<td>DB</td>
<td>3</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>861</td>
<td>PL</td>
<td>2</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>861</td>
<td>AI</td>
<td>2</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>955</td>
<td>PL</td>
<td>2</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>955</td>
<td>AI</td>
<td>2</td>
<td>96</td>
</tr>
</tbody>
</table>

Why do we need both WHERE and HAVING?

Attention testing
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
```

Aggregate Functions

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

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
```

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.name
```
Task (for the end of this part)

<table>
<thead>
<tr>
<th>Student name</th>
<th>Faculty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alma</td>
<td>CS</td>
</tr>
<tr>
<td>Amir</td>
<td>CS</td>
</tr>
<tr>
<td>Ahuva</td>
<td>EE</td>
</tr>
</tbody>
</table>

What is the average #likes per posting in each faculty? Show only for faculties with >2 liked postings.

Pairs / groups allowed. Email solution to me: bennyk@cs.technion.ac.il

Outline

- Introduction
- Basic SQL Queries
- Aggregation and Grouping
  - NULLs
- Nested SQL Queries
- Views

Missing Information

- Problem: pieces of data missing, but we need to keep whatever partial knowledge we have.

SQL’s 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)

Try More Yourself (psql)

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');

Enroll student course
Ahuva PL
Alon

Course course lecturer
PL Eran

\( \bot \)

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

Of course, we’ve lost our initial association (join)...
Conditions with NULL

- Principle: atomic predicates (e.g., comparison between two numbers) result in unknown (denoted \( U \)) when one or more operands is NULL.
  - \( 5 > \text{NULL} \), \( \text{NULL} = \text{NULL} \), 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 \land y )</th>
<th>( x \lor y )</th>
<th>( \neg x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
<td>( F )</td>
</tr>
<tr>
<td>( T )</td>
<td>( U )</td>
<td>( U )</td>
<td>( U )</td>
<td>( U )</td>
</tr>
<tr>
<td>( U )</td>
<td>( T )</td>
<td>( U )</td>
<td>( T )</td>
<td>( T )</td>
</tr>
<tr>
<td>( U )</td>
<td>( U )</td>
<td>( U )</td>
<td>( U )</td>
<td>( U )</td>
</tr>
<tr>
<td>( F )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
<td>( F )</td>
</tr>
<tr>
<td>( F )</td>
<td>( U )</td>
<td>( U )</td>
<td>( U )</td>
<td>( F )</td>
</tr>
<tr>
<td>( F )</td>
<td>( F )</td>
<td>( F )</td>
<td>( F )</td>
<td>( F )</td>
</tr>
</tbody>
</table>

Can You Explain It Now?

Avoiding Nulls

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

DDL: constrain on non-nullity

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

Queries: nullity testing

Outer Joins

Left Outer Join

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

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

We’ve lost Amir!
Definition

- Let \( r \) and \( s \) be relations over schemas \( R(A_1, ..., A_n) \) and \( S(B_1, ..., B_m) \), respectively.
- We define:
  \[ r \bowtie s = (r \times s) \cup \{(r \times A_{(n+1)} \ldots n}) \times \{(\perp, \ldots, \perp)\}\]  
  
- That is, \( r \bowtie s \) 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 in SQL

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

Full Outer Join

```sql
r \bowtie s = (r \bowtie s) \cup (s \bowtie s)
```

Left Outer Join in SQL

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

Right Outer Join in SQL

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

Full Outer Join in SQL

```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 $Nx1$
- We denote a query that returns a single column and at most one row as $1x1$
- Subqueries (nested queries) are sometimes required to be $Nx1$ or $1x1$
  - In PostgreSQL, $1x1$ is checked at runtime, hence, this property is sensitive to the database
  - As opposed to $Nx1$
- To denote that there is no restriction we will use $NxM$

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

```
<table>
<thead>
<tr>
<th>Student</th>
<th>Took</th>
<th>Course</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>course</td>
<td>grade</td>
</tr>
<tr>
<td>861</td>
<td>DB</td>
<td>99</td>
</tr>
<tr>
<td>753</td>
<td>PL</td>
<td>82</td>
</tr>
<tr>
<td>955</td>
<td>AI</td>
<td>72</td>
</tr>
</tbody>
</table>
```

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)
```

Refined Example

```
<table>
<thead>
<tr>
<th>Student</th>
<th>Took</th>
<th>Course</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>course</td>
<td>grade</td>
</tr>
<tr>
<td>861</td>
<td>DB</td>
<td>99</td>
</tr>
<tr>
<td>753</td>
<td>PL</td>
<td>82</td>
</tr>
<tr>
<td>955</td>
<td>AI</td>
<td>72</td>
</tr>
</tbody>
</table>
```

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

```
SELECT S.name
FROM Student S, Took T, Course C
WHERE S.sid=T.sid AND T.course=C.course AND C.credit=2
```
The FROM Scope

• 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

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)
```

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)
```

Example of IN

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 of NOT IN

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'
)
```

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 with NOT EXISTS

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'
)
```
Nesting Inside SELECT

• 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

Student | Task | Course
----- | ---- | ----
| sid | name | year | course | grade | course | credit |

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

Student | Task | Course
----- | ---- | ----
| sid | name | year | course | grade | course | credit |

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

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

Outline

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

Practical Problem Related to Example

StudentCourse | Student | Task | Course
----- | ---- | ---- | ----
| sid | name | year | course | grade | course | credit |

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 ahova`
  - I can live with her seeing who took what, but not the grades!

Solution: View

Definition: A view is a stored query that can be accessed as an ordinary relation

```
CREATE VIEW STC as
  SELECT sid, course FROM Took WHERE ...
GRANT SELECT on STC to ahova
```

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