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
  - 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 to RA

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

Except that RA does not produce duplicates

Basic SQL Query

- \textbf{SELECT} attributes \textbf{FROM} relations \textbf{WHERE} condition

Example:

\[
\text{SELECT course} \\
\text{FROM Student, Enroll} \\
\text{WHERE Student.sid = Enroll.sid}
\]

<table>
<thead>
<tr>
<th>Student</th>
<th>Enroll</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>name</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>sid</td>
<td>course</td>
</tr>
</tbody>
</table>
DISTINCT Eliminates Duplicate Tuples

$$\pi_{A_1 \ldots A_k}^{\text{Condition}(B_1 \ldots B_m)}(R_1 \times \ldots \times R_n)$$

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

Background: 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 $C = A \cup B$, $\xi = \mu + \lambda$
  - $(A, \mu) \cap (B, \lambda) = (C, \xi)$ where $C = A \cap B$, $\xi = \min(\mu, \lambda)$
  - $(A, \mu) \setminus (B, \lambda) = (C, \xi)$ where $C = A \setminus B$, $\xi = \max(0, \mu - \lambda)$

Discussion

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

Example

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

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

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

Example

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

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

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

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

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

11/20/17
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>Amir</td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
</tr>
</tbody>
</table>

Example with DISTINCT

<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>Amir</td>
</tr>
<tr>
<td>955</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

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

Example: Attribute Naming (Aliasing)

```
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
```

How Come?

```
\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:

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

where \( t \) is a tuple in \( \sigma_{\text{Condition}(B_1,\ldots,B_m)}(R_1\times\ldots\times R_n) \)

Question

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

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?
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?
(No conditions capture at most one one row in the Cartesian product)

What Does This Query Return?

```
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)
    =, !=, >, >=, <=, between(x AND y)
  - Membership in lists:
    IN(x1,...,xk), NOT IN(x1,...,xk)
- (Later: EXISTS, > ANY, > ALL, IS NULL, ...)

Example 1

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

Example 2

```
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)?
Set Operations

- We can apply union, intersection and difference to two (or more) queries
  - \((Q_1 \cup Q_2)\)
  - \((Q_1 \cap Q_2)\)
  - \((Q_1 \setminus Q_2)\)
- Subqueries must be union compatible in a *weak sense*
  - Same number of attributes
  - Types must be convertible to each other (e.g., int → float)
  - Output adopts the schema of the first 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
  - \((Q_1 \cup \text{ALL} \ Q_2)\)
  - \((Q_1 \cap \text{ALL} \ Q_2)\)
  - \((Q_1 \setminus \text{ALL} \ Q_2)\)

Question Revisited

<table>
<thead>
<tr>
<th>Student</th>
<th>(SELECT sid FROM Student) UNION (SELECT id FROM Employee)</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>Amir</td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Employee</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
</tr>
<tr>
<td>233</td>
</tr>
<tr>
<td>651</td>
</tr>
<tr>
<td>122</td>
</tr>
</tbody>
</table>

What are the Results?

- \((\text{SELECT name FROM Student}) \cup \text{ALL} (\text{SELECT name FROM Employee})\)
- \((\text{SELECT name FROM Student}) \cap \text{ALL} (\text{SELECT name FROM Employee})\)
- \((\text{SELECT name FROM Student}) \setminus \text{ALL} (\text{SELECT name FROM Employee})\)

Answer A2 (last page)

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

ORDER BY

- \(\text{SELECT } A_1, \ldots, A_n \text{ FROM } R_1, \ldots, R_m \text{ WHERE } \text{Condition}(B_1, \ldots, B_n) \text{ ORDER BY } C_1, \ldots, C_k\)
  - \(C_1, \ldots, C_k\) are all in \(R_1, \ldots, R_m\)

Example:

- \(\text{SELECT } * \text{ FROM } \text{Student, Enroll WHERE } \text{Student.sid} = \text{Enroll.sid ORDER BY course, name}\)
### Example 1

<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>Amir</td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
</tr>
</tbody>
</table>

**SQL**

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

### Example 2

<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>Amir</td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
</tr>
</tbody>
</table>

**SQL**

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

### Example 3

<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>Amir</td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
</tr>
</tbody>
</table>

**SQL**

```sql
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.

```sql
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:

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

### 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
  - \( \text{ROUND}(v, i) \): round \( v \) to \( i \) decimals
  - \( \text{UPPER}(v) \): convert string to uppercase
  - \( -v, v+w, v\times w \)... 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

- **Scalar functions over aggregates**
  - \( 2\times\text{AVG}(C) - \text{AVG}(C1) + \text{AVG}(C2) / \exp(\text{SUM}(C)) \)

**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>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>

**Columns**

- What counts as a **column**?
  - Attribute name from \( R_1, \ldots, R_n \)
    - course, grade, ...
  - Scalar functions over attribute names
    - \( \text{grade/2}, \text{grade*credit}, \text{UPPER(name)} \)
    - Semantics: create a new column by computing the value row by row
  - Duplicate elimination: \( \text{DISTINCT C} \)
    - \( \text{DISTINCT UPPER(name)} \)

**Aggregate Query**

```
Aggregate function over a column

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

**Semantics of Aggregate Queries**

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

where each \( C_i \) is a **column** obtained from \( \sigma_{\text{Condition}(B_1, \ldots, B_m)}(R_1 \times \ldots \times R_n) \)

**Example**

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

<table>
<thead>
<tr>
<th>num</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>91</td>
</tr>
</tbody>
</table>
### How Many Courses?

<table>
<thead>
<tr>
<th>id</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>

- **SELECT COUNT(course) as num FROM Took**

- **SELECT COUNT(DISTINCT course) as num FROM Took**

- **SELECT COUNT(DISTINCT UPPER(course)) as num FROM Took**

### Average Per Course

<table>
<thead>
<tr>
<th>course</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Databases</td>
<td>75</td>
</tr>
<tr>
<td>PL</td>
<td>76</td>
</tr>
</tbody>
</table>

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

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

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

Problem with this solution?

### Group Selection

**SELECT agg(8) GROUP BY A HAVING A>a**

**SELECT agg(8) GROUP BY A**

### Group Selection (binary group)

**SELECT agg(8) GROUP BY A, A2 HAVING A2=f**

**SELECT agg(8) GROUP BY A, A2**

Two tuples belong to the same group if they are equal on all grouping attributes.
Grouping Syntax

SELECT Values(G1,...,Gl,agg1,...,aggq) FROM R1,...,Rn
WHERE Condition(B1,...,Bm) GROUP BY G1,...,Gl
HAVING Condition(G1,...,Gl,agg1,...,aggq)

Two tuples belong to the same group if they are equal on all of G1,...,Gl.

Obtained only from grouping attributes and aggregates over the other attributes.

Why do we need both WHERE and HAVING?

Answer A4 (last page)

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

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

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

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
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 ...
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 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;
```

### Task

<table>
<thead>
<tr>
<th>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>

### Solution

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

```
SELECT S.faculty, 
    count(*) * 1.0/count(DISTINCT(L.posting)) as average
FROM Likes L, Posting P, Student S
WHERE L.posting = P.id AND P.owner = S.name
GROUP BY S.faculty
HAVING count(DISTINCT(L.posting)) > 2;
```

### 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

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

```
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>PL</td>
</tr>
<tr>
<td></td>
<td>Keren</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>Keren</td>
<td></td>
</tr>
</tbody>
</table>
```
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)

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');

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

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

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

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 &amp; Y</th>
<th>X \lor Y</th>
<th>¬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>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>U</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>U</td>
<td>T</td>
<td>U</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>U</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>U</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

Try It 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, lecturer
FROM Enroll R, Course C
WHERE R.course = C.course;

Can You Explain It Now?

What problem of 3VL does this example show?

Any suggestion for an alternative semantics?

(Attend advanced course)
Avoiding Nulls

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

DDL: constrain on non-nullity

SELECT sid
FROM Enroll
WHERE course IS NOT NULL

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

Queries: nullity testing

Left Outer Join

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>753</td>
<td>Nazareth</td>
</tr>
<tr>
<td>852</td>
<td>Alek</td>
</tr>
</tbody>
</table>

Right Outer Join

\[
\text{r⟕s} = (\text{r} \bowtie \text{s}) \cup \left( \left( \prod_{B_1,\ldots,B_n} (\text{r} \bowtie \text{s}) \right) \times \left( \bot, \ldots, \bot \right) \right)
\]

Full Outer Join

\[
\text{r} \bowtie \text{s} = (\text{r} \bowtie \text{s}) \cup (\text{r} \bowtie \text{s})
\]
11/20/17

**Left Outer Join in SQL**

<table>
<thead>
<tr>
<th>Student</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSN</td>
<td>name</td>
</tr>
<tr>
<td>861</td>
<td>Alma</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
</tr>
<tr>
<td>852</td>
<td>Ahuva</td>
</tr>
</tbody>
</table>

**Generalized Semantics in SQL (1)**

- Every \( t \) in \( R \times S \) such that \( \text{Cond}(t) = \text{true} \)
- Every \( r \) in \( R \) such that \( \text{Cond}(t) = \text{false} \) for all \( t \) in \( R \times S \)
  - \( r \) extended with nulls

**Generalized Semantics in SQL (2)**

- Every \( t \) in \( R \times S \) such that \( \text{Cond}(t) = \text{true} \)
- Every \( r \) in \( R \) such that \( \text{Cond}(t) = \text{false} \) for all \( t \) in \( R \times S \)
  - \( r \) extended with nulls

**Right Outer Join in SQL**

<table>
<thead>
<tr>
<th>Student</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSN</td>
<td>name</td>
</tr>
<tr>
<td>861</td>
<td>Alma</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
</tr>
<tr>
<td>852</td>
<td>Ahuva</td>
</tr>
</tbody>
</table>

**Full Outer Join in SQL**

<table>
<thead>
<tr>
<th>Student</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSN</td>
<td>name</td>
</tr>
<tr>
<td>861</td>
<td>Alma</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
</tr>
<tr>
<td>852</td>
<td>Ahuva</td>
</tr>
</tbody>
</table>

**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

Subquery in WHERE

- Most common place for subqueries
- Several forms:
  - As any scalar value (\(1x1\))
    - \(T\).\texttt{grade} \(\geq\) (SELECT MAX(\texttt{grade}) FROM \texttt{Took})
    - Empty result is treated as NULL
  - Membership testing (\(N\times1\))
    - \(T\).\texttt{course} \texttt{IN} (SELECT \texttt{name} FROM \texttt{Course} WHERE \texttt{credit}=2)
  - Nonemptiness testing (\(N\timesM\))
    - \texttt{EXISTS} (SELECT \* FROM \texttt{Course} WHERE \texttt{credit}=2)

Example

Task: Find the students who got the maximal grades

\[
\text{SELECT S.name FROM Student S, Took T WHERE S.sid=T.sid AND T.grade >= (SELECT MAX(grade) from Took)}
\]

Refined Example

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

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

Previous example:

\[
\begin{align*}
\text{SELECT} & \quad S.\text{name} \\
\text{FROM} & \quad \text{Student} \ S, \ \text{Took} \ T \\
\text{WHERE} & \quad S.\text{sid}=T.\text{sid} \ \text{AND} \\
T.\text{grade} & \quad \geq (\text{SELECT} \ \text{MAX}(\text{grade}) \ \text{FROM} \ \text{Took})
\end{align*}
\]

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

\[
\begin{align*}
\text{SELECT} & \quad S.\text{name} \\
\text{FROM} & \quad \text{Student} \ S, \ \text{Took} \ T \\
\text{WHERE} & \quad S.\text{sid}=T.\text{sid} \ \text{AND} \\
T.\text{grade} & \quad \geq (\text{SELECT} \ \text{MAX}(\text{grade}) \ \text{FROM} \ \text{Took})
\end{align*}
\]

Example with EXISTS

Task: Find the students who attended courses that Alma took

\[
\begin{align*}
\text{SELECT} & \quad S.\text{name} \\
\text{FROM} & \quad \text{Student} \ S, \ \text{Took} \ T \\
\text{WHERE} & \quad S.\text{sid}=T.\text{sid} \ \text{AND} \\
\text{EXISTS} & \quad (\text{SELECT} * \\
\text{FROM} & \quad \text{Student, Took} \\
\text{WHERE} & \quad \text{Student.}s\text{id}=\text{Took.}s\text{id} \ \text{AND} \\
& \quad \text{course} = T.\text{course} \ \text{and} \ \text{name} = 'Alma')
\end{align*}
\]

Example with NOT EXISTS

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

\[
\begin{align*}
\text{SELECT} & \quad S.\text{name} \\
\text{FROM} & \quad \text{Student} \ S, \ \text{Took} \ T \\
\text{WHERE} & \quad S.\text{sid}=T.\text{sid} \ \text{AND} \\
\text{NOT EXISTS} & \quad (\text{SELECT} * \\
\text{FROM} & \quad \text{Student, Took} \\
\text{WHERE} & \quad \text{Student.}s\text{id}=\text{Took.}s\text{id} \ \text{AND} \\
& \quad \text{course} = T.\text{course} \ \text{and} \ \text{name} = 'Alma')
\end{align*}
\]

Example of IN

Task: Find the students who attended courses with >100 students

\[
\begin{align*}
\text{SELECT} & \quad S.\text{name} \\
\text{FROM} & \quad \text{Student} \ S, \ \text{Took} \ T \\
\text{WHERE} & \quad S.\text{sid}=T.\text{sid} \ \text{AND} \\
T.\text{course} & \quad \text{IN} (\text{SELECT} \ \text{course} \\
& \quad \text{FROM} \ \text{Took} \\
\text{GROUP} & \quad \text{BY} \ \text{course} \\
\text{HAVING} & \quad \text{COUNT}(*)>100)
\end{align*}
\]

Example of NOT IN

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

\[
\begin{align*}
\text{SELECT} & \quad S.\text{name} \\
\text{FROM} & \quad \text{Student} \ S, \ \text{Took} \ T \\
\text{WHERE} & \quad S.\text{sid}=T.\text{sid} \ \text{AND} \\
T.\text{course} & \quad \text{NOT IN} (\text{SELECT} \ T1.\text{course} \\
& \quad \text{FROM} \ \text{Student} \ S1, \ \text{Took} \ T1 \\
\text{WHERE} & \quad S1.\text{sid}=T1.\text{sid} \ \text{AND} \\
& \quad S1.\text{name} = 'Alma')
\end{align*}
\]

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

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;
```

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 S.sid NOT IN (SELECT sid FROM Took WHERE course=C.course);
```

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

```
StudentCourse
```

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!

Outline

- Introduction
- Basic SQL Queries
- Aggregation and Grouping
- NULLs
- Nested SQL Queries
- Views
Solution: View

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

- Incremental view maintenance is an active technological and research topic
  - Another fascinating topic: updating the database by updating the view (a.k.a. view updates)

Answers to Questions

- **A1**: all students (id,name) who enrolled in >1 courses
  - DISTINCT matters; otherwise we get the pair for each (E,F)
- **A2**:
  1. Alma, Amir, Ahuva, Avia, Avi
  2. Alma, Amir, Ahuva, Alma, Avia, Avi
  3. Alma, Amir, Ahuva
- **A3**: The probability that all dice turn heads
- **A4**: WHERE filters rows from the Cartesian product (e.g., age>20) while HAVING filters whole groups (e.g., count>5)