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
  - Stroustrup’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)
```

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

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

Basic SQL to RA

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

Except that RA does not produce duplicates
DISTINCT Eliminates Duplicate Tuples

Eliminate duplicate tuples

\[
\text{SELECT (DISTINCT } A_1, \ldots, A_k \text{ FROM } R_1, \ldots, R_n \text{ WHERE Condition}(B_1, \ldots, B_m) \]

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

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 \(\mu : A \to \mathbb{N}\) 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 \(\xi = \mu + \lambda\)
  - \((A, \mu) \cap (B, \lambda) = (C, \xi)\) where \(C = A \cap B\) and \(\xi = \min(\mu, \lambda)\)
  - \((A, \mu) \setminus (B, \lambda) = (C, \xi)\) where \(C = A\) and \(\xi = \max(0, \mu - \lambda)\)

Question

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>861</td>
<td>Alma</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>955</td>
<td>Ahuva</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

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

Example

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

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

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

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

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

```
<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>
```

Example with DISTINCT

```
<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>
```

```
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)

Practice: [http://sqlfiddle.com](http://sqlfiddle.com)

```
create table Student
(sid int, name varchar, year int);
create table Enroll
(sid int, course varchar);
```

```
insert
into Student
values
(861,'Alma',2), (753,'Amir',1), (955,'Ahuva',2);
```

```
insert
into Enroll
values
(861,'DB'), (861,'PL'), (753,'PL');
```

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

```
<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>
```

Example: Select All Attributes

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

```
<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>year</th>
<th>enrol.sid</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
<td>861</td>
<td>DB</td>
</tr>
<tr>
<td>861</td>
<td>PL</td>
<td></td>
<td>861</td>
<td>PL</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
<td>753</td>
<td>PL</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>enrol.sid</th>
<th>name</th>
<th>year</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>Alma</td>
<td>2</td>
<td>DB</td>
</tr>
<tr>
<td>861</td>
<td>PL</td>
<td></td>
<td>PL</td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td>1</td>
<td>PL</td>
</tr>
</tbody>
</table>
Example: Attribute Naming (Aliasing)

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

Example: Functions on Attributes

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

How Come?

SQL deploys a generalized model:

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

Relation Naming (Aliasing)

```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'
```

What Does This Query Return?

```sql
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
```
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, ...)

Example 1

```sql
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>

Example 2

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

<table>
<thead>
<tr>
<th>sid</th>
<th>course</th>
<th>credit</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<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>

Question

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

Bag or Set Semantics?

- We can apply union, intersection and difference to two (or more) queries
  - \((Q_1) \text{ UNION } (Q_2)\)
  - \((Q_1) \text{ INTERSECT } (Q_2)\)
  - \((Q_1) \text{ EXCEPT } (Q_2)\)
- Subqueries must be union compatible in a weak sense
  - Same #attributes
  - Types must be convertible to each other (e.g., int \(\rightarrow\) float)
  - Output adopts the schema of 1st subquery
- 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

```
(SELECT sid FROM Student)
UNION
(SELECT id FROM Employee)
```

What are the Results?

```
(SELECT name FROM Student)
UNION
(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

ORDER BY

```
SELECT A_1,...,A_k
FROM R_1,...,R_n
WHERE Condition(B_1,...,B_k)
ORDER BY C_1,...,C_l
```

Example:

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

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 ASC, course DESC
```
**Example 3**

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

<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>

<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>AI</td>
</tr>
<tr>
<td>753</td>
<td>DC</td>
</tr>
</tbody>
</table>

**Top-k Tuples**

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

```
SELECT A1,...,Ak
FROM R1,...,Rn
WHERE Condition(B1,...,Bm)
ORDER BY C1,...,Ck
LIMIT k
```

Example:

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

**Outline**

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

**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

**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, v1, v2, ...`: 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

\[
\text{SELECT } \text{Agg}_{1}(C_{1}), \ldots, \text{Agg}_{k}(C_{k}) \text{ FROM } R_{1}, \ldots, R_{n} \text{ WHERE Condition}(B_{1}, \ldots, B_{m})
\]

Example:

\[
\text{SELECT } \text{SUM}(\text{credit}) \text{ FROM Took \ WHERE } \text{sid}=861
\]

Semantics of Aggregate Queries

where each \( C_{i} \) is a column obtained from \( \sigma_{\text{Condition}(B_{1}, \ldots, B_{m})}(R_{1} \times \ldots \times R_{n}) \)

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, \( \text{UPPER(name)} \)
  - Duplicate elimination: \( \text{DISTINCT C} \)
    - \( \text{DISTINCT UPPER(name)} \)

Example

\[
\text{SELECT COUNT(\text{sid}) as } \text{num} \text{ FROM Took}
\]

\[
\text{SELECT COUNT(\text{distinct course}) as } \text{num} \text{ FROM Took}
\]

How Many Courses?

\[
\text{SELECT COUNT(course) as } \text{num} \text{ FROM Took}
\]

\[
\text{SELECT COUNT(\text{distinct course}) as } \text{num} \text{ FROM Took}
\]

\[
\text{SELECT COUNT(\text{distinct UPPER(course)}) as } \text{num} \text{ FROM Took}
\]

What Does This Compute?

\[
\text{SELECT exp(\text{sum(ln(prob))}) FROM UnbalancedDice}
\]
### Average Per Course

<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>91</td>
</tr>
<tr>
<td>861</td>
<td>PL</td>
<td>2</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>

### Group Selection

**Example**

<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>752</td>
<td>AI</td>
<td>93</td>
</tr>
<tr>
<td>955</td>
<td>AI</td>
<td>72</td>
</tr>
<tr>
<td>699</td>
<td>AI</td>
<td>96</td>
</tr>
<tr>
<td>729</td>
<td>AI</td>
<td>87</td>
</tr>
</tbody>
</table>

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

\[
\text{average} = \frac{79 \times 3 + 87 \times 2 + 93}{5} = 81.2
\]

### Aggregate Functions

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

\[
\text{SELECT} \quad \text{avg(grade)}
\]

\[
\text{FROM} \quad \text{Student} S, \text{Course} C
\]

\[
\text{WHERE} \quad S.\text{sid} = T.\text{sid} \text{ AND} \ T.\text{course} = C.\text{name}
\]
### Aggregate Functions

**Task:** Find the average grade of 2nd 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
HAVING S.year = 2 AND SUM(C.credit) >= 5;
```

**Question**

What is the result?

```
create table S (l varchar, r int);
insert into S values ('A',0), ('A',1), ('B',2), ('B',3), ('C',4), ('D',0), ('D',2);
```

```
SELECT l FROM S WHERE r > 1 GROUP BY l HAVING count(1) > 1
```
**Another Question**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>D</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
<td>D</td>
<td>2</td>
</tr>
</tbody>
</table>

SELECT S.l FROM S, T WHERE S.l = T.l AND S.r > 1 GROUP BY S.l HAVING count(*) > 1

What is the result?

**Task**

<table>
<thead>
<tr>
<th>Name</th>
<th>Faculty</th>
<th>Post</th>
<th>Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alma</td>
<td>CS</td>
<td>23</td>
<td>Alma</td>
</tr>
<tr>
<td>Amir</td>
<td>CS</td>
<td>45</td>
<td>Amir</td>
</tr>
<tr>
<td>Ahuva</td>
<td>EE</td>
<td>76</td>
<td>Ahuva</td>
</tr>
<tr>
<td></td>
<td></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.

**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?

```
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 E, Course C
WHERE E.course = C.course;
```

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

**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...

```
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 E, Course C
WHERE E.course = C.course;
```

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

**Try It Yourself (PostgreSQL)**
Try More Yourself (psql)

<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>

| SELECT student FROM Enroll WHERE course='PL'; |
| SELECT student FROM Enroll WHERE course='PL'; |
| SELECT student FROM Enroll WHERE course=PL' OR course='PL'; |

Conditions with NULL

- Principle: atomic predicates (e.g., comparison between two numbers) result in unknown (denoted U) when one or more operands is NULL
  - S=NULL NULL=NULL, etc.
- Then, propositional logic over atomic predicates follows the three-valued logic (3VL):

Three-Valued Logic (3VL)

Can You Explain It Now?

<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>

| SELECT student FROM Enroll WHERE course='PL'; |
| SELECT student FROM Enroll WHERE course='PL'; |
| SELECT student FROM Enroll WHERE course=PL' OR course='PL'; |

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

Can You Explain It Now?

<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>

| SELECT student FROM Enroll WHERE course='PL'; |
| SELECT student FROM Enroll WHERE course='PL'; |
| SELECT student FROM Enroll WHERE course=PL' OR course='PL'; |

What problem of 3VL does this example show?
Any suggestion for an alternative semantics?
(Attend advanced course)

Outer Joins

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

<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>

<table>
<thead>
<tr>
<th>Student</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>

We've lost Amir!
Left Outer Join

<table>
<thead>
<tr>
<th>SSN</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>851</td>
<td>Ahuva</td>
<td></td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td></td>
</tr>
<tr>
<td>852</td>
<td>Ahuva</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SSN</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>851</td>
<td>Haifa</td>
</tr>
<tr>
<td>955</td>
<td>Tiberias</td>
</tr>
<tr>
<td>852</td>
<td>Acre</td>
</tr>
</tbody>
</table>

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

Student

<table>
<thead>
<tr>
<th>SSN</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>851</td>
<td>Ahuva</td>
<td></td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td></td>
</tr>
<tr>
<td>852</td>
<td>Ahuva</td>
<td></td>
</tr>
</tbody>
</table>

Address

<table>
<thead>
<tr>
<th>SSN</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>851</td>
<td>Haifa</td>
</tr>
<tr>
<td>955</td>
<td>Tiberias</td>
</tr>
<tr>
<td>852</td>
<td>Acre</td>
</tr>
</tbody>
</table>

Right Outer Join

<table>
<thead>
<tr>
<th>SSN</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>851</td>
<td>Ahuva</td>
<td></td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td></td>
</tr>
<tr>
<td>852</td>
<td>Ahuva</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SSN</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>851</td>
<td>Haifa</td>
</tr>
<tr>
<td>955</td>
<td>Tiberias</td>
</tr>
<tr>
<td>852</td>
<td>Acre</td>
</tr>
</tbody>
</table>

Full Outer Join

Student

<table>
<thead>
<tr>
<th>SSN</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>851</td>
<td>Ahuva</td>
<td></td>
</tr>
<tr>
<td>753</td>
<td>Amir</td>
<td></td>
</tr>
<tr>
<td>852</td>
<td>Ahuva</td>
<td></td>
</tr>
</tbody>
</table>

Address

<table>
<thead>
<tr>
<th>SSN</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>851</td>
<td>Haifa</td>
</tr>
<tr>
<td>955</td>
<td>Tiberias</td>
</tr>
<tr>
<td>852</td>
<td>Acre</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 $\text{Cond}(r) = \text{false}$ for all $t$ in $R \times S$
  - ($r$ extended with nulls)

**R LEFT JOIN S ON (Cond)**

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

**R RIGHT JOIN S ON (Cond)**

- Every $t$ in $R \times S$ such that $\text{Cond}(t) = \text{true}$
- Every $s$ in $S$ such that $\text{Cond}(t) = \text{false}$ for all $t$ in $R \times S$
  - ($s$ extended with nulls)
Generalized Semantics in SQL(2)

\[ R \text{ FULL JOIN } S \text{ ON } (\text{Cond}) \]

- 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 \( R \times S \)
- Every \( s \) in \( S \) such that \( \text{Cond}(t) = \text{false} \) for all \( t \) in \( R \times S \)

(R and S 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>

\[ \text{SELECT A.SSN, name, year, city} \]
\[ \text{FROM Person S RIGHT JOIN Address A ON (S.SSN = A.SSN)} \]

<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>

Student

<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>302</td>
<td></td>
</tr>
</tbody>
</table>

Full Outer Join in SQL

\[ \text{SELECT * FROM Person S FULL JOIN Address A ON (S.SSN = A.SSN)} \]

<table>
<thead>
<tr>
<th>S.SSN</th>
<th>A.SSN</th>
<th>name</th>
<th>year</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>861</td>
<td>861</td>
<td>Alma</td>
<td>2</td>
<td>Haifa</td>
</tr>
<tr>
<td>852</td>
<td>852</td>
<td>Ahuva</td>
<td>2</td>
<td>Afek</td>
</tr>
<tr>
<td>753</td>
<td>753</td>
<td>Amir</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>302</td>
<td>302</td>
<td></td>
<td></td>
<td>Nazareth</td>
</tr>
<tr>
<td>955</td>
<td>955</td>
<td></td>
<td></td>
<td>Isfiya</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

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)

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

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)

Next...
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

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 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

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

```sql
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)
```

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!

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
```

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>Update queries</td>
<td>Faster</td>
<td>Slower</td>
</tr>
<tr>
<td>No extra update overhead</td>
<td>No extra storage overhead</td>
<td>Slower updates</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)