MSiA-413 Introduction to Databases and Information Retrieval

Lecture 15
Common Table Expressions (CTEs)
Recursive queries on networks and hierarchies
Views, existential operators, set comparison

Instructor: Nikos Hardavellas

Slides adapted from S. Tarzia, A. Silberschatz, H.F. Korth, S. Sudarshan

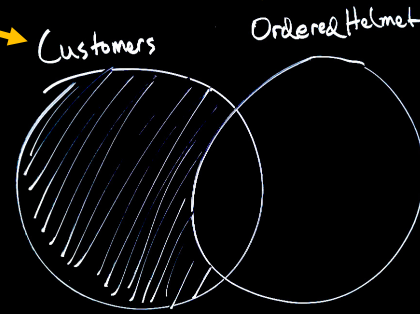
Last Lecture

- `UNION`, `INTERSECT` clauses
  - Similar to set operations
- `CASE` statements
  - Similar to *if* … *then* … *else* programming language constructs
- Introduced Regular Expressions
  - Used to *search for text* matching a complex *pattern*
  - It is often a trial-and-error process to find the right regular expression

Ordinary Common Table Expressions: `WITH`

`SalesOrders.sqlite`: Which customers never ordered a helmet?

```
SELECT CustomerID
  FROM Customers
  LEFT JOIN
    (SELECT DISTINCT CustomerID AS helmet_customer
     FROM Orders NATURAL JOIN Order_Details NATURAL JOIN Products
     WHERE ProductName LIKE "%Helmet%")
  ON CustomerID = helmet_customer
WHERE helmet_customer IS NULL;
```



… is equivalent to …

Create temp table "OrderedHelmet"

Populate "OrderedHelmet"

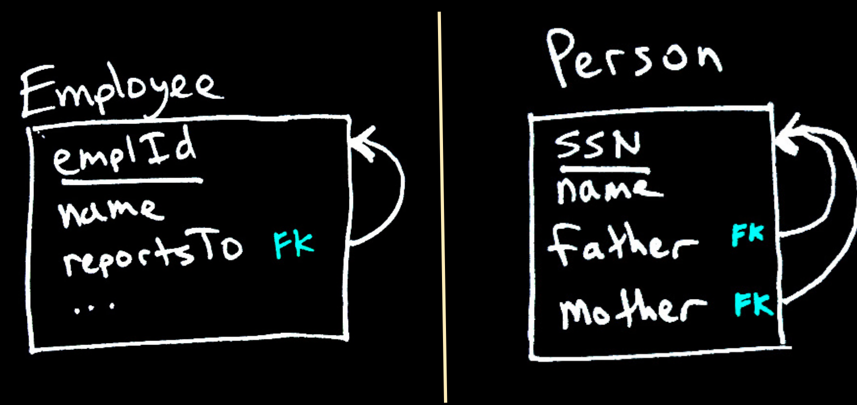
```
WITH OrderedHelmet(helmet_customer) AS
    (SELECT DISTINCT CustomerID
     FROM Orders NATURAL JOIN Order_Details NATURAL JOIN Products
     WHERE ProductName LIKE "%Helmet%")
SELECT CustomerID
  FROM Customers
  LEFT JOIN OrderedHelmet
    ON CustomerID = helmet_customer
WHERE helmet_customer IS NULL;
```

Query "OrderedHelmet"

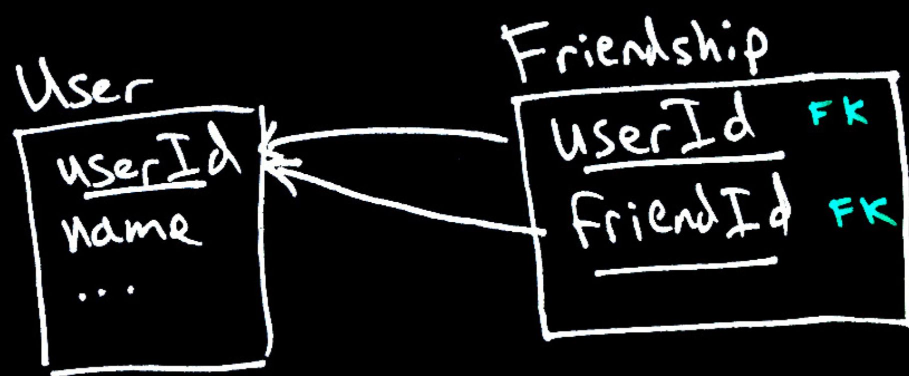
Drop "OrderedHelmet" when done

Hierarchies and Networks

- Occur when:
  - A table has a foreign key referring to the same table (many-to-one):



  - A linking table has foreign keys referring to the same table (many-to-many):



- *Recursion* allows complex structures to be represented with a set of simple relationships

SQL difficulties with networks

- SQL relational databases can model networks, but it's difficult to write queries that traverse the network
- For example, "find all of the classes that must be taken before ACC 257"
  - Prerequisite of ACC 257 is ACC 220
  - Prerequisite of ACC 220 is ACC 210
  - ACC 210 has no prerequisites
- Must do a JOIN or subquery every time you take a step in the network
  - May need to do this many, many times!

"Find all classes that must be taken before ACC 257"

`SchoolScheduling.sqlite`

Output of each step is listed as a new column

Two steps in the network

```
SELECT s2.SubjectCode, s3.SubjectCode
 FROM Subjects AS s1
     JOIN Subjects AS s2 ON s2.SubjectCode=s1.SubjectPrereq
     JOIN Subjects AS s3 ON s3.SubjectCode=s2.SubjectPrereq
 WHERE s1.SubjectCode="ACC 257";
```

Output is one wide row:

	SubjectCode	SubjectCode
1	ACC 220	ACC 210

This "plain SQL" approach is not scalable

- Query must grow to accommodate additional steps in the transitive relationship
- If we had 5 subjects as prereqs, then we would need 5 joins and 5 output columns
- How many joins to write? (we may not know beforehand how many prereqs there are)

Traversing networks with multiple queries

Use a general-purpose programming language (like Python or R)

- Generate a sequence of SQL commands to traverse the network
- This can be inefficient if the SQL server is remote

Pseudo-python:

```
prereqs = []
subjCode = "ACC 257"
while subjCode:
    queryResult = db.query(
         "SELECT SubjectPrereq FROM Subjects WHERE SubjectCode = " + subjCode)
    subjCode = queryResult.getRow(0).getColumn(0)
    if subjCode != None:
        prereqs.append(subjCode)
return prereqs
```

We want to execute SQL iterations like this:

- Create a temporary table

```
create temp table prereq(SubjectCode);
```

- Iteration 1: find prereq of 257

```
insert into prereq
values('ACC 257');
```

```
SELECT SubjectPrereq
FROM Subjects
NATURAL JOIN prereq WHERE
Subjects.SubjectCode='ACC 257';
```

Output: ACC 220

- Iteration 2: find prereq of 220

```
insert into prereq
values('ACC 220');
```

```
SELECT SubjectPrereq
FROM Subjects
NATURAL JOIN prereq WHERE
Subjects.SubjectCode='ACC 220';
```

Output: ACC 210

- Iteration 3: find prereq of 210

```
insert into prereq
values('ACC 210');
```

```
SELECT SubjectPrereq
FROM Subjects
NATURAL JOIN prereq WHERE
Subjects.SubjectCode='ACC 210';
```

Output: NULL (this is 210's column value)

- Iteration 4: find prereq of NULL

```
insert into prereq
values(NULL);
```

```
SELECT SubjectPrereq
FROM Subjects
NATURAL JOIN prereq WHERE
Subjects.SubjectCode=NULL;
```

Output: empty (no output)

	subjCode
1	ACC 257
2	ACC 220
3	ACC 210
4	*NULL*

- Terminate. Get output

```
SELECT * FROM prereq;
```

Recursive Common Table Expressions example

- A way to express a chain of dependent queries

```
WITH RECURSIVE
  prereq(SubjectCode)
  AS (
    VALUES('ACC 257')
    UNION
    SELECT SubjectPrereq
      FROM Subjects
         NATURAL JOIN prereq
  )
SELECT * FROM prereq;
```

Create a temporary table "prereq" with one column named "subjCode"

Values of first row

Contents of "prereq" table

Recursive query to get more rows from each ***new*** *prereq row*

When you finish the recursion, here is the output I want

Output:

	subjCode
1	ACC 257
2	ACC 220
3	ACC 210
4	NULL

Recursive Common Table Expressions

```
WITH RECURSIVE
  tmp_table(column_name1, column_name2, …)
  AS (
    non_recursive_initial_SELECT
    UNION [ALL]
    recursive_SELECT
) final_SELECT
```

temporary table to store results

compound `SELECT` query (without `ORDER BY`, `LIMIT`, `OFFSET`) or constant for the initial rows of the results

ALL optionally includes duplicate rows

`SELECT` query that refers to table tmp_table and generates new rows; tmp_table must be referenced exactly once here and nowhere else; no aggregates or window functions; LIMIT limits #rows added to the table total in the recursive step; ORDER BY retrieves rows from queue in order

final `SELECT` query that retrieves data from table tmp_table

Mechanics (see `https://sqlite.org/lang_with.html`):

1. Create an empty temporary table matching the defined schema
2. Run the first query to generate initial rows and add these to a queue
3. While the queue is not empty, remove a row from the queue
   a) Add the row to the temporary table
   b) Run the recursive `SELECT` using just this one row to represent the temporary table
   c) Add results to the queue and repeat step 3
4. Run the final `SELECT` query, using the temporary table generated above

Set Comparison: SOME and ANY

Find the instructors with salary greater than that of some (at least one) instructor in the Biology department

```
SELECT T.name
FROM instructor AS T, instructor AS S
WHERE T.salary > S.salary AND S.dept_name = 'Biology';
```

- Same query using **SOME** clause

```
SELECT name
FROM instructor
WHERE salary > SOME (SELECT salary
                     FROM instructor
                     WHERE dept_name = 'Biology');
```

- **SOME** and **ANY** are equivalent. Neither is supported by SQLite.

Set Comparison: ALL

Find the instructors with salary greater than that of all instructors in the Biology department

```
SELECT name
FROM instructor
WHERE salary > ALL (SELECT salary
                    FROM instructor
                    WHERE dept_name = 'Biology’);
```

- ALL is not supported by SQLite

Test for empty relations: EXISTS

Find the employees with salary greater than that of a manager's

```
SELECT name
FROM employee
WHERE salary > SOME (SELECT salary
                     FROM manager);
```

- Same query using **EXISTS** clause

```
SELECT name
FROM employee AS E
WHERE EXISTS (SELECT salary
              FROM manager AS M
              WHERE E.salary > M.salary);
```

- The query becomes mathematical: $\{e.name \mid e \in E \land \exists m \in M{:}\, e.salary > m.salary\}$
- The **NOT EXISTS** clause does the opposite

Views

- Sometimes, it is not desirable for all users to see the entire logical model
- Say, we want to allow users to know who the rich agents are, but we do not want to reveal their actual salary

```
SELECT Agentid, AgtFirstName, AgtLastName
FROM Agents WHERE Salary > 30000;
```

AgentID	AgtFirstName	AgtLastName
1	William	Thompson
6	John	Kennedy

- A **view** provides a mechanism to achieve this
- Any relation that is not of the conceptual model, but it is made visible to a user as a "virtual relation" is called a **view**

View definition and use

- A view definition causes the saving of an expression

```
CREATE VIEW rich_agents AS
  SELECT Agentid, AgtFirstName, AgtLastName
  FROM Agents
  WHERE Salary > 30000;
```

- The expression is substituted into queries that use the view

```
SELECT * FROM rich_agents;
```

AgentID	AgtFirstName	AgtLastName
1	William	Thompson
6	John	Kennedy

- Is equivalent to

```
SELECT * FROM (SELECT Agentid, AgtFirstName, AgtLastName
               FROM Agents
               WHERE Salary > 30000);
```

- ...but without revealing the salary value used in the conditional
- The `WITH` clause defines a temporary view with scope limited to the query