>>

SQL Queries (v): Abstraction

∧ >>

❖ Complex Queries

For complex queries, it is often useful to

- break the query into a collection of smaller queries
- define the top-level query in terms of these

This can be accomplished in several ways in SQL:

- views (discussed in detail below)
- subqueries in the **WHERE** clause
- subqueries in the **FROM** clause
- subqueries in a **WITH** clause

VIEWs and **WHERE** clause subqueries haveen discussed elsewhere.

WHERE clause subqueries can be correlated with the top-level query.

<< ∧ >>

❖ Complex Queries (cont)

Example: get a list of low-scoring students in each course
(low-scoring = mark is less than average mark for class)

Schema: *Enrolment(course,student,mark)*

Approach:

- generate tuples containing *(course,student,mark,classAvg)*
- select just those tuples satisfying *(mark < classAvg)*

Implementation of first step via window function

```
SELECT course, student, mark,
       avg(mark) OVER (PARTITION BY course)
FROM   Enrolments;
```

split all tuples into partitions based on the course code, the calculate the average mark for each partition.

We now look at several ways to complete this data request ...

Note here that the avg(mark) aggregate is still part of the select statement, just because it is on a new-line doesn't mean it is separate. The from keyword is the next keyword, and it hasn't been seen by that point yet.

that is why the resulting tuple has form (course, stu, mark, avg)

Why a partition, and not a group? We need the avg mark per student, not just per course, and a partition is used when we want to use the aggregate value for each tuple of a "group". GROUP BY results in 1 tuple per group, partition results in multiple tuples per group, to each the aggregate is tacked on to.

❖ Using Views for Abstraction

Defining complex queries using views:

```
CREATE VIEW
   CourseMarksWithAvg(course,student,mark,avg)
AS
SELECT course, student, mark,
       avg(mark) OVER (PARTITION BY course)
FROM   Enrolments;

SELECT course, student, mark
FROM   CourseMarksWithAvg
WHERE  mark < avg;
```

<< ^ >>

❖ Using Views for Abstraction (cont)

In the general case:

```
CREATE VIEW View1(a,b,c,d) AS Query1;
CREATE VIEW View2(e,f,g) AS Query2;
...
SELECT attributes
FROM   View1, View2
WHERE  conditions on attributes of View1 and View2
```

Notes:

- look like tables ("virtual" tables)
- exist as objects in the database (stored queries)
- useful if specific query is required frequently

<< ∧ >>

❖ FROM-clause Subqueries for Abstraction

Defining complex queries using **FROM** subqueries:

```
SELECT course, student, mark
FROM   (SELECT course, student, mark,
               avg(mark) OVER (PARTITION BY course)
        FROM   Enrolments) AS CourseMarksWithAvg
WHERE  mark < avg;
```

the as CourseMarksWithAvg doesn't actually do anything but provide an understandable alias name so that when you come back to this query you can quickly tell what the from query is doing without having to re-read and re-understand it, you instead just read the alias name

NOTE, providing the AS name alias is actually mandatory for sub-queries

Avoids the need to define views.

<< ∧ >>

❖ FROM-clause Subqueries for Abstraction (cont)

In the general case:

```
SELECT attributes
FROM   (Query1) AS Name1,
       (Query2) AS Name2
       ...
WHERE  conditions on attributes of Name1 and Name2
```

Notes:

- must provide name for each subquery, even if never used
- subquery table inherits attribute names from query
  (e.g. in the above, we assume that *Query*$_1$ returns an attribute called **a**)

❖ WITH-clause Subqueries for Abstraction

Defining complex queries using **WITH**:

```
WITH CourseMarksWithAvg AS
     (SELECT course, student, mark,
             avg(mark) OVER (PARTITION BY course)
      FROM   Enrolments)
SELECT course, student, mark, avg
FROM   CourseMarksWithAvg
WHERE  mark < avg;
```

Avoids the need to define views.

❖ WITH-clause Subqueries for Abstraction (cont)

In the general case:

```
WITH   Name1(a,b,c) AS (Query1),
       Name2 AS (Query2), ...
SELECT attributes
FROM   Name1, Name2, ...
WHERE  conditions on attributes of Name1 and Name2
```

Notes:

- *$Name_1$*, etc. are like temporary tables
- named tables inherit attribute names from query

<< ^ >>

❖ Recursive Queries

`WITH` also provides the basis for recursive queries.

Recursive queries are structured as:

```
WITH RECURSIVE R(attributes) AS (
      SELECT ... not involving R
    UNION
      SELECT ... FROM R, ...
)
SELECT attributes
FROM   R, ...
WHERE  condition involving R's attributes
```

Useful for scenarios in which we need to traverse multi-level relationships.

❖ Recursive Queries (cont)

For a definition like

```
WITH RECURSIVE R AS ( Q1 UNION Q2 )
```

Q_1 does not include **R** (base case); **Q_2** includes **R** (recursive case)

How recursion works:

```
Working = Result = evaluate Q1
while (Working table is not empty) {
    Temp = evaluate Q2, using Working in place of R
    Temp = Temp - Result
    Result = Result UNION Temp
    Working = Temp
}
```

i.e. generate new tuples until we see nothing not already seen.

<< ∧

❖ Recursive Queries (cont)

Example: count numbers of all sub-parts in a given part.

Schema: *Parts(part, sub_part, quantity)*

```
WITH RECURSIVE IncludedParts(sub_part, part, quantity) AS (
    SELECT sub_part, part, quantity
    FROM   Parts WHERE part = GivenPart
  UNION ALL
    SELECT p.sub_part, p.part, p.quantity
    FROM   IncludedParts i, Parts p
    WHERE  p.part = i.sub_part
  )
SELECT sub_part, SUM(quantity) as total_quantity
FROM   IncludedParts
GROUP  BY sub_part
```

Includes sub-parts, sub-sub-parts, sub-sub-sub-parts, etc.

Produced: 5 Oct 2020