

# SQL Queries (v): Abstraction

---

- Complex Queries
- Using Views for Abstraction
- **FROM**-clause Subqueries for Abstraction
- **WITH**-clause Subqueries for Abstraction
- Recursive Queries

## ❖ 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 have been 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;
```

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

## ❖ 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;
```

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*<sub>1</sub> 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*<sub>1</sub>, 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<sub>1</sub>** does not include **R** (base case); **Q<sub>2</sub>** 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: 27 Feb 2021