SQL Functions and Expressions

Expressions in SELECT clause

- Expressions may occur in a SELECT clause

  SELECT expression [AS column name] (, ...)

- The result is a relation with a single tuple

SELECT 1 AS one;

one
1

SELECT sqrt(2), 'John '||`Smith', 2=3;

sqrt		
1.4142	John Smith	f

Expressions in SELECT clause returns a relation

- Expressions may occur in a SELECT clause

  SELECT expression [AS column name] (, ...)
- The result is a relation with a single tuple
- We can therefore place such an expression in a FROM clause of another query

```
SELECT   q.one
FROM     (SELECT 1 AS one) q
```

one
1

Using a SELECT expression inside other expressions

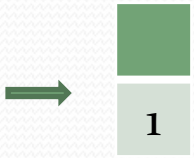
- To use a SELECT expression statement as a value in another expression, it is required to place parentheses around that expression

- The following is incorrect:

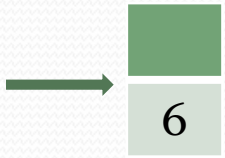
```
SELECT SELECT 1
```

- The following statements are correct:

```
SELECT (SELECT 1)
```




```
SELECT (SELECT 2)*(SELECT 3)
```

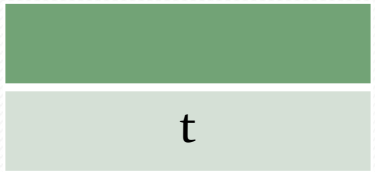


Application: Boolean queries

- A boolean query is a query that returns either true (t) or false (f).
- E.g., "Are there CS students who take courses?"

```
SELECT EXISTS(SELECT  S.Sid
              FROM    Student S, Enroll E
              WHERE S.Major = 'CS' AND S.Sid = E.Sid);
```

- If yes, then answer is



- If no, the answer is

Application: Checking for keys

- "Is Major a primary key for the Student relation?"
- I.e., if Major where a primary key, then there can not exist two different students with the same major.
- We can check this with the Boolean query

```
SELECT NOT EXISTS(SELECT  *
                  FROM    Student S1, Student S2
                  WHERE S1.Sid <> S2.Sid AND
                        S1.Major = S2.Major) AS IsKey;
```

Application: Checking for Foreign keys

- "Is Sid a foreign key in the Enroll relation referencing the primary key Sid in the Student relation?"
- I.e., is it true that there does not exist a Sid value in the Enroll relation that does not appear as a Sid value in the Student relation?

```
SELECT NOT EXISTS (SELECT  E.Sid
                   FROM    Enroll E
                   WHERE E.Sid NOT IN (SELECT S.Sid
                                       FROM    Student S)) AS isFK;
```

Employee

Eid	Salary
p1	10000
p2	10000
p3	20000

- "Find new employee salaries after an increase by 5%."

```
SELECT E.Eid, E.Salary*1.05 AS NewSalary
FROM   Employee E
```

Eid	NewSalary
p1	10500
p2	10500
p3	21000

Eid	Salary
p1	10000
p2	10000
p3	20000

- "Report whether or not an employee has the lowest salary."

```
SELECT E.Eid,  E.Salary ≤ ALL(SELECT E1.Salary
                              FROM   Employee E1) AS
                    HasLowestSalary
FROM Employee E
```

Eid	HasLowestSalary
p1	t
p2	t
p3	f

CASE expression

```
SELECT E.Eid,
        CASE WHEN E.Salary > 100000  THEN 'high'
             WHEN E.Salary < 10000   THEN 'low'
             ELSE 'medium'
         END AS SalaryRange
FROM Employee E;
```

Eid	Salary
p1	25000
p2	9000
p3	105000

Eid	SalaryRange
p1	medium
p2	low
p3	high

Expressions with parameterized sub-queries

- "Report whether or not a student takes a course in the department in which he or she majors."

```
SELECT S.Sid, S.Major IN (SELECT C.Dept
                          FROM   Course C, Enroll E
                          WHERE E.Sid = S.Sid AND E.Cno = C.Cno)
FROM   Student S
```

- Notice how S.Sid is a parameter in the subquery
- For S.Sid, the subquery returns the departments that offer courses in which the student with that sid is enrolled

Expressions in WHERE clause

- Consider a relation of points in 2-dimensional space.

Pid	X	Y
p1	0	0
p2	1	0
p3	2	1
p4	5	3

- "Find the pairs of points that are within distance 3."

```
SELECT P1.Pid AS P1, P2.Pid AS P2
FROM   Point P1, Point P2
WHERE sqrt(power(P1.X-P2.X,2)+power(P1.Y-P2.Y,2)) ≤ 3
```

- sqrt(.) and power(.,.) are PostgreSQL functions.

Putting things together

- "Raise the salary of an employee by 5% provided that the raise is less than $1000."

```
(SELECT  E.Eid, E.Salary*1.05 AS NewSalary
  FROM    Employee E
  WHERE E.Salary * 0.05 <  1000)
UNION
(SELECT  E.Eid, E.Salary AS NewSalary
  FROM    Employee E
  WHERE E.Salary  * 0.05  ≥ 1000)
```

Eid	Salary
p1	10000
p2	10000
p3	20000

Eid	NewSalary
p1	10500
p2	10500
p3	20000

Application: polynomials

- Consider the polynomial $3x^3 - 5x + 7$
- We can represent such a polynomial in a relation Polynomial(coefficient int, degree int).

Polynomial

coefficient	degree
3	3
-5	1
7	0

Derivative of polynomial

- If ax^n is a term of a polynomial and $n \geq 1$ then its derivative is $n * ax^{n-1}$
- The derivative of the previous polynomial is $3 * 3x^2 + 1 * (-5) = 9x^2 - 5$
- We can compute this derivative with an SQL query:

```
SELECT t.degree*t.coefficient AS coefficient, t.degree-1 AS degree
FROM   Polynomial t
WHERE t.degree >= 1
```

coefficient	degree
9	2
-5	0

User-defined functions

- Users can define there own functions and then use them in expressions.
- "Define the increment-by 1 function."

```
CREATE FUNCTION increment (n INTEGER)
  RETURNS INTEGER AS
  $$
     SELECT n+1;
  $$ LANGUAGE SQL;
```

- This function can now be used in a query. E.g.,

```
SELECT increment(5) AS Value;
```

Value
6

User-defined functions

- Write the distance function between two points (x_1, y_1) and (x_2, y_2): $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

```
CREATE FUNCTION distance(x1 FLOAT, y1 FLOAT, x2 FLOAT, y2 FLOAT)
    RETURNS FLOAT AS
    $$
        SELECT sqrt(power(x1-x2,2)+power(y1-y2,2));
    $$  LANGUAGE SQL;
```

Pid	X	Y
p1	0	0
p2	1	0
p3	2	1
p4	5	3

- "Find the pairs of points that are within distance 3."

```
SELECT P1.Pid AS P1, P2.Pid AS P2
FROM   Point P1, Point P2
WHERE distance(P1.X,P1.Y,P2.X,P2.Y) ≤ 3
```

Functions with output parameters

- Functions can take input parameters
- Functions can also have output parameters (OUT)
- OUT parameters are useful to return tuples and relations with several attributes named by these parameters.

```
CREATE FUNCTION sum_and_product(x int, y int, OUT sum int, OUT product int)
AS
$$
    SELECT x+y, x*y;
$$  LANGUAGE SQL;
```

- Example,

```
SELECT t.sum, t.product
FROM    sum_and_product(3,4) t;
```

sum	product
7	12

Functions with output parameters return tuples

```
CREATE FUNCTION sum_and_product(x int, y int, OUT sum int, OUT product int)
AS
$$
    SELECT x+y, x*y;
$$  LANGUAGE SQL;
```

```
SELECT sum_and_product(3,4);
```

→

sum_and_product
(7,12)

```
SELECT (sum_and_product(3,4)).sum, (sum_and_product(3,4)).product
```

→

sum	product
7	12

Record (tuple) types

- SQL permits the definition of record (tuple) types.
- Functions can return a record.

```
CREATE TYPE edge AS (source INT, target INT);

CREATE FUNCTION printEdge(x INT, y INT) RETURNS edge
AS $$
        SELECT x, y;
    $$ LANGUAGE SQL;
```

```
SELECT * FROM printEdge(1,2);
```

→

source	target
1	2

```
SELECT source FROM printEdge(1,2);
```

→

source
1

Functions returning sets (relations)

- A function can also return a set of tuples (relation).
- The return type of such as function is specified using the RETURN SET record clause.
- Let Pair(x int, y int) be a relation of pairs:

```
CREATE FUNCTION sum_and_product(OUT sum int, OUT product int)
RETURNS SETOF RECORD
AS $$
    SELECT P.x+P.y, P.x*P.y FROM Pair P;
$$ LANGUAGE SQL;
```

- Can be used for user-defined views.

Alternative RETURNS TABLE

```
CREATE FUNCTION sum_and_product()
RETURNS TABLE (sum INTEGER, product INTEGER)
AS $$
    SELECT P.x+P.y, P.x*P.y FROM Pair P;
$$ LANGUAGE SQL;
```

Pair

x	y
1	2
3	-4

```
SELECT t.sum, t.product
FROM    sum_and_product() t;
```

sum	product
3	2
-1	-12

Caution: non determinism

- Functions returning a record return a single record even if the body of the function computes a set of record.
- Leads to non-deterministic effects.

```
CREATE FUNCTION sum_and_product(OUT sum int, OUT product int)
RETURNS RECORD
AS $$
    SELECT P.x+P.y, P.x*P.y FROM Pair P ORDER BY RANDOM();
$$ LANGUAGE SQL;

SELECT * FROM sum_and_product();
```

sum	product
3	2

or

sum	product
-1	-12