EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Advanced SQL
Sommersemester 2022
Prof. Torsten Grust, Christian Duta
WSI — Database Systems Research Group

Assignment 5 (03.06.2022)

Handin until: 17.06.2022, 09:00

Until June 15th 2022, students have the opportunity to **evaluate lectures**. Please help us to improve **your** courses by providing precious feedback. Check your Mailbox **now** to participate.

1. [15 Points] **Array Representations**

   Arrays can be represented in different ways. One is to use the built-in arrays of PostgreSQL. As such, we populate the following table s:

```sql
CREATE TABLE s (
  arr_id integer PRIMARY KEY,
  arr    text[]
);
```

```sql
INSERT INTO s VALUES
(1, ARRAY['a','b','c']),
(2, ARRAY['d','d']);
```

   Now, let us assume that there is **no built-in array data type**. We would then encode arrays in regular tables using explicit element positions (see Chapter 4, slide 4). This leads us to table **t** which replaces table s:

```sql
CREATE TABLE t (
  arr_id integer,
  idx    integer,
  val    text,
  PRIMARY KEY(arr_id, idx)
);
```

```sql
INSERT INTO t VALUES
(1,1,'a'),(1,2,'b'),(1,3,'c'),
(2,1,'d'),(2,2,'d');
```

   Likewise, queries that used to rely on built-in array operations (see the five queries (a) - (e) below which refer to table s) would need to be rewritten into queries over table **t** without any reference to such operations. Queries that originally returned array values would now return their tabular encodings instead.

   For example, we expect the following result when (d) is rewritten assuming the sample instances of table s and t above:

arr_id	idx	val
1	1	a
1	2	b
1	3	c
1	4	e
1	5	f
2	1	d
2	2	d
2	3	e
2	4	f

Rewrite the queries (a) - (e) below such that the rewritten queries reference table `t` instead of table `s` and do not exhibit any array functions and operators[1]. The rewritten queries must be semantically equivalent.

(a)

```
SELECT s.arr[1] AS val
FROM   s AS s
WHERE  s.arr_id = 1;
```

(b)

```
SELECT s.arr_id,
       array_length(s.arr,1) AS len
FROM   s AS s;
```

(c)

```
SELECT s.arr_id, a AS val
FROM   s             AS s,
       unnest(s.arr) AS a;
```

(d)

```
SELECT s.arr_id,
       array_cat(s.arr,ARRAY['e','f'])
FROM   s AS s;
```

(e)

```
TABLE s
  UNION ALL
SELECT new.id           AS arr_id,
       s.arr||'g'::text AS arr
FROM s AS s, (
  SELECT MAX(s.arr_id) + 1
  FROM   s AS s
) AS new(id)
WHERE s.arr_id = 1;
```

2. [5 Points] **Transpose Two-Dimensional Arrays**

We provide you with a table definition `matrices` with two-dimensional arrays (matrices). Assume that every matrix in this table has the same dimensions.

```
CREATE TABLE matrices (
  matrix text[][] NOT NULL
);
```

Write a SQL query which transposes each matrix.

Example:

```
INSERT INTO matrices VALUES
(array[['1','2','3'],
       ['4','5','6']]),
(array[['f','e','d'],
       ['c','b','a']]);
```

transposed
{{1,4},{2,5},{3,6}}
{{f,c},{e,b},{d,a}}

[1] https://www.postgresql.org/docs/current/functions-array.html

3. [10 Points] **Array Tree**

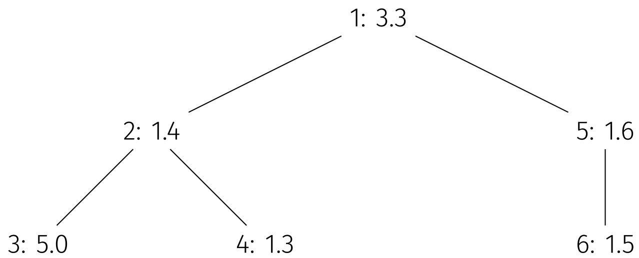
We introduced the possibility to represent trees in terms of two arrays representing parent and label information (see Chapter 4, Slide 6). For this assignment, we use `numeric` labels to define a table `trees` of array-encoded trees:

```
CREATE TABLE trees (
  tree    int PRIMARY KEY,
  parents int[],
  labels  numeric[]
);
```

Example: Populate the table `trees` with some sample trees.

```
INSERT INTO trees VALUES
(1, ARRAY[NULL,1,2,2,1,5],
    ARRAY[3.3,1.4,5.0,1.3,1.6,1.5]),
(2, ARRAY[3,3,NULL,3,2],
    ARRAY[0.4,0.4,0.2,0.1,7.0]);
```

Drawing `tree 1` would result in the following graph ($n : l$ indicates that node n has label l):



Write a SQL query which, for each node n, calculates the label sum of n's immediate children. For the example above the result would be:

tree	node	sum
1	1	3.0
1	2	6.3
1	3	0.0
1	4	0.0
1	5	1.5
1	6	0.0
2	1	0.0
2	2	7.0
2	3	0.9
2	4	0.0
2	5	0.0