EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Advanced SQL
Winter Term 2025/2026
Prof. Torsten Grust, Tim Fischer, Björn Bamberg
WSI — Database Systems Research Group

Assignment 6

Hand in this assignment until **Friday, December 5th 2025, 12:00 pm** at the latest.

🤔 Running out of ideas?
Are you hitting a roadblock? Are some of the exercises unclear? Do you just need that one hint to get the ball rolling? Refer to the `#forum` 🌐 channel on our Discord server—maybe you'll find just the help you need.

⚠️ The lecture evaluation is coming up!
We rely on your feedback to steer Advanced SQL in the right direction. We look forward to both your criticism and your praise! So please take part in the lecture evaluation by December 5th 2025. Thank you very much!

Ⓔxam-style Exercises
Exercises marked with Ⓔ are similar in style to those you will find in the exam. You can use these to hone your expectations and gauge your skills.

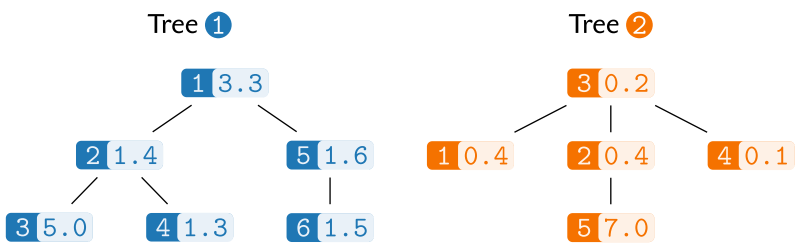
Task 1: List Tree

LISTS TREES LIST ENCODING

In the lecture, we discussed how to represent trees using a pair of lists, one encoding the parents of each node and the other the labels. Table `trees` uses this encoding to hold multiple trees with `numeric` labels, each identified by column `tree`. Once drawn, the two trees in the example instance of `trees` to the right look as follows:

`trees`

tree	parents	labels
①	[NULL, 1, 2, 2, 1, 5]	[3.3, 1.4, 5.0, 1.3, 1.6, 1.5]
②	[3, 3, NULL, 3, 2]	[0.4, 0.4, 0.2, 0.1, 7.0]



Write a SQL query which, for each tree in `trees`, computes the sums of each nodes immediate children.

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

😵‍💫 Don't know where to start?
To get started, consider defining a macro `list_positions(lst, el)`, which returns *all* indices of the element `el` in the list `lst` (similar to the built-in function `list_position(lst, el)` 🌐). The macro should return the empty list if `el` is not in `lst`.

The built-in function `generate_series` 🌐 may also come in handy. We will be discussing it in the coming lecture—in short, its a table-valued function that *generates* a *series* of numbers.

🤔 Is it working?
For the example instance of `trees` given above, your query should give the output on the right.

Task 2: Transpose Two-Dimensional Lists

LISTS MULTIDIMENSIONAL LISTS

The `list` data type in DuckDB also supports nested, or multidimensional, lists. We can use this multidimensionality is to, among other things, represent matrices, *e.g.*, the data type `text[][]` can represent matrices of strings.

Consider table `matrices` to the right, which contains multiple matrices, each identified by an `id`.

`matrices`

id	matrix
1	[['1','2','3'], ['4','5','6']]
2	[['l','k','j','i'], ['h','g','f','e'], ['d','c','b','a']]

`output`

id	transposed
1	[['1','4'], ['2','5'], ['3','6']]
2	[['l','h','d'], ['k','g','c'], ['j','f','b'], ['i','e','a']]

Write a SQL query which transposes each `matrix`. For the example instance of ⊞ `matrices` above, the output of your query should look like ⊞ `output`.

🤝 Need a hand?
`WITH ORDINALITY` and/or the `generate_series` macro from the lecture make solving this task a breeze. 😉

☝ Don't be lazy!
Your query has to work for any possible instance of ⊞ `matrices`, not just the specific instance shown here! Its okay if fails for non-matrix entries, but it has to work for every legal `matrix`.

Task 3: Tabular Lists Ⓔ

LISTS ARRAYS TABULAR ENCODING

DuckDB offers `list`s as a built-in data type to work with ordered collections—we can store them in tables, we can use them in queries, *etc.* Put briefly, they are a useful tool to have! But do we really need them? Do we loose the ability to express certain data or queries over it if you we don't have a dedicated `list` data type?

⊞ s

lst_id	lst
1	['a','b','c']
2	['d','d']

As we discussed in the chapter 4 (slides 4f) of the lecture: ***we don't need a dedicated*** `list` ***type!*** At least when we don't take convenience into account… Consider table ⊞ `s`, it holds multiple `list`s each differentiated by a given `lst_id`. We can construct table ⊞ `t`, containing exactly the same information as ⊞ `s`: it holds multiple lists, again differentiated by `lst_id`, and each of these lists holds multiple values `val` at explicit positions `idx`.

⊞ t

lst_id	idx	val
1	1	'a'
1	2	'b'
1	3	'c'
2	1	'd'
2	2	'd'

Rewrite the following queries such that they **only make use of the tabular encoding of lists** —*i.e.*, use ⊞ `t` instead of ⊞ `s` and replace any `list`-specific functions and operators 🌐 with semantically equivalent SQL.

A
```sql
SELECT s.lst[1] AS val
FROM   s AS s
WHERE  s.lst_id = 1;
```

B
```sql
SELECT s.lst_id,
       len(s.lst) AS len
FROM   s AS s;
```

C
```sql
SELECT s.lst_id, val
FROM   s                AS s,
       unnest(s.lst) AS _(val);
```

D
```sql
SELECT s.lst_id,
       s.lst || ['e','f']
FROM   s AS s;
```

E
```sql
TABLE s
  UNION ALL
SELECT new.id                       AS lst_id,
          list_append(s.lst, 'g') AS lst
FROM   s AS s, (
          SELECT max(s.lst_id) + 1
          FROM   s AS s
       ) AS new(id)
WHERE  s.lst_id = 1;
```

💡 Unsure if you are the right track?
Take a look at the following results you should see after rewriting the respective queries.

Subtask A

val
'a'

Subtask B

lst_id	len
1	3
2	2

Subtask C

lst_id	val
1	'a'
1	'b'
1	'c'
2	'd'
2	'd'

Subtask D

lst_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'

Subtask E

lst_id	idx	val
1	1	'a'
1	2	'b'
1	3	'c'
2	1	'd'
2	2	'd'
3	1	'a'
3	2	'b'
3	3	'c'
3	4	'g'