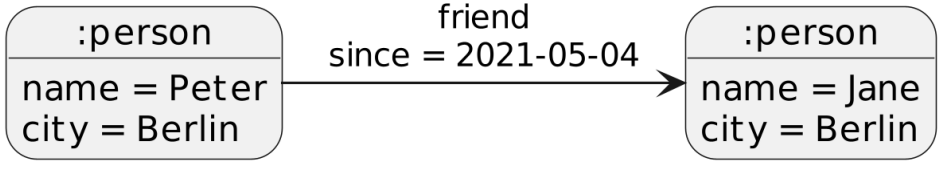
Neo4J (Graph Database)

Data Model: Property Graph $G = (V, E)$

- *Vertices*: have a label (node type) and a set of properties (key-value pairs)
  `(:person { "name": "Peter", "city": "Berlin" })`

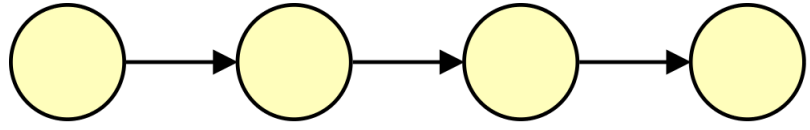
- *Edges*: directed connection between two nodes, have a label and properties
  `() -[:friend {"since":"2021-05-04"}]-> ()`



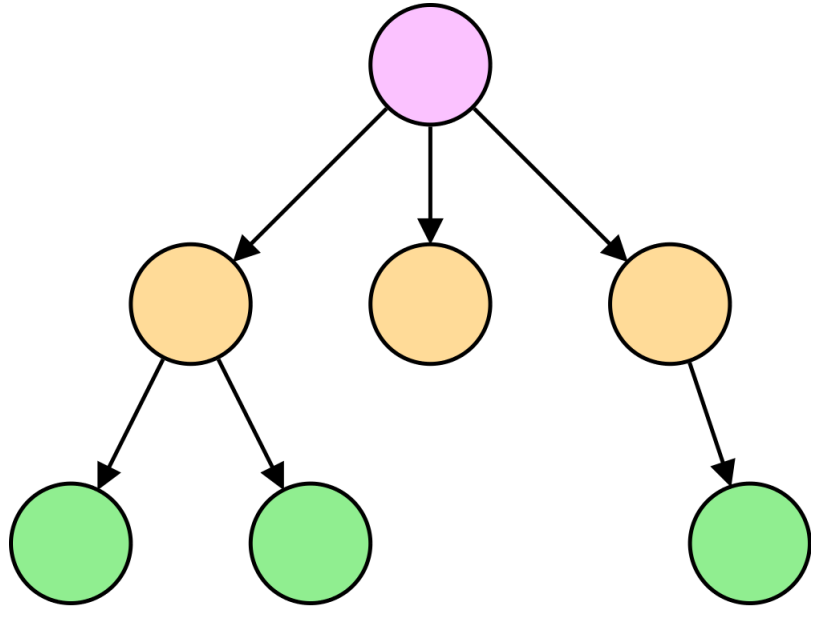
Graphs consist of nodes edges. In typical graph databases, both nodes and edges can have a label, and a list of arbitrary properties. In Neo4J, there are only directed edges. So, when we want to model a friendship between two person nodes - friendship is a symmetric relationship -, we simply choose an arbitrary direction, either an outgoing or incoming edge. Later, when querying the data, we simply traverse edges in both directions to find a person's friends.

Graph types

List: Each node has at most one parent and child

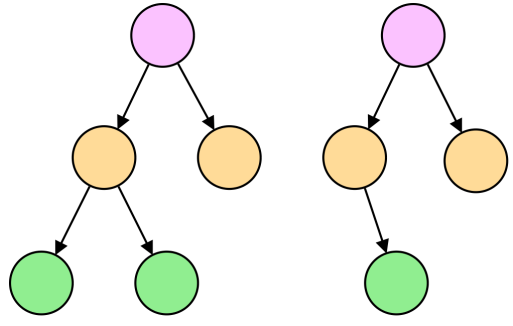


(Directed) Tree: Each node has at most one parent

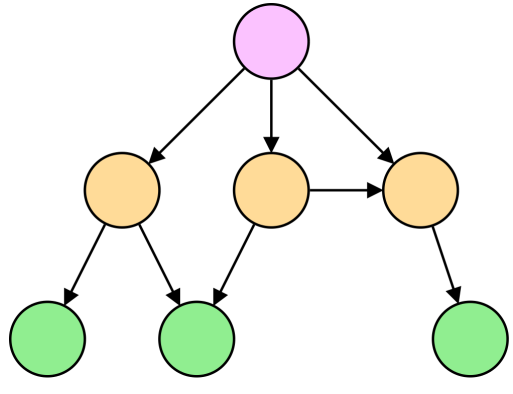


Graph types

(Directed) Forest: Multiple disjoint trees

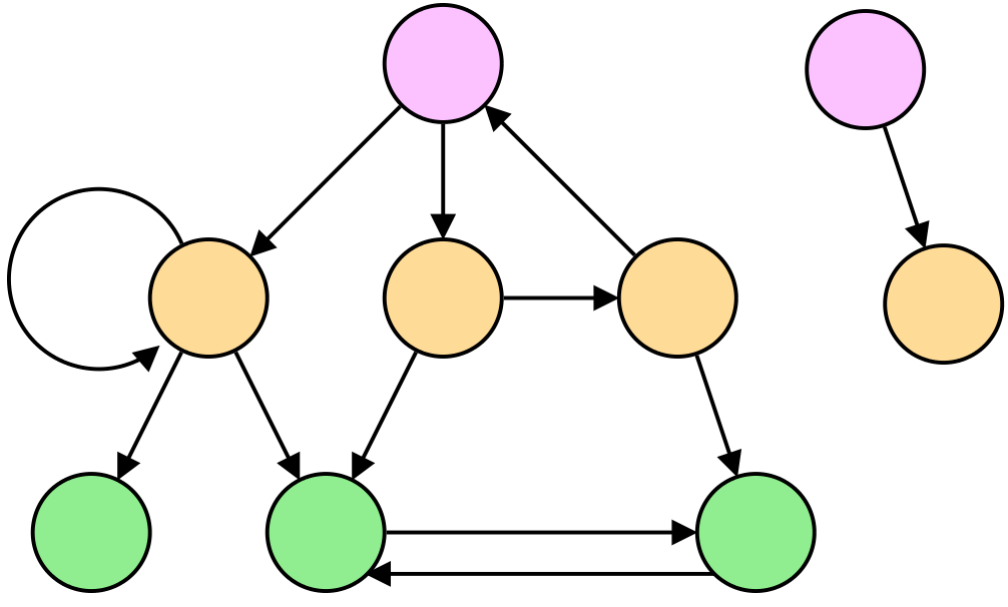


DAG: A Directed Acyclic Graph

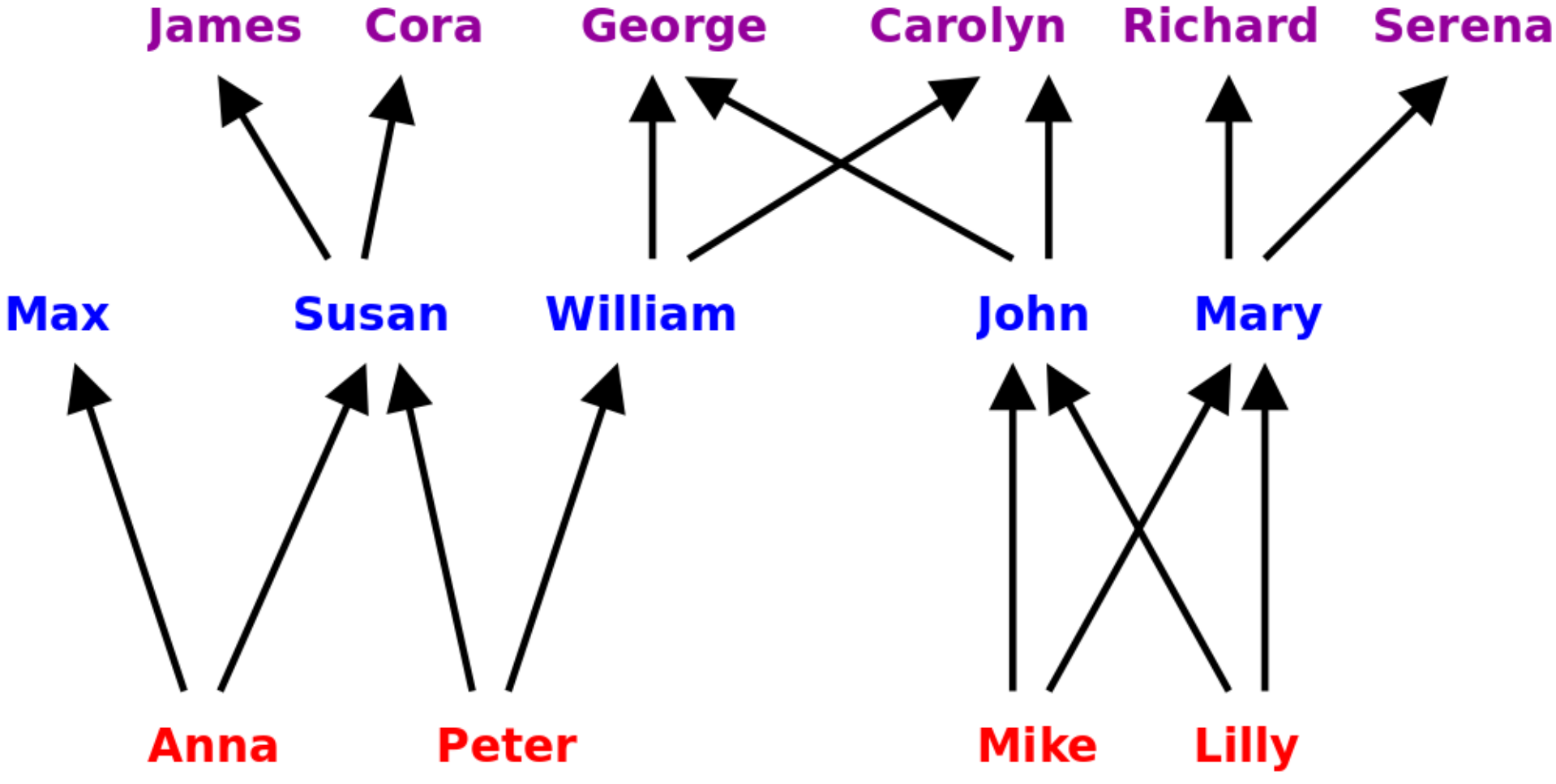


Graph types

General graph



Example: Genealogy tree



Neo4J (Graph Database)

Some typical graph operations:

- Find connections (of specific types) from a node to other nodes
- Find connections of degree **n**
- Find shortest path between two nodes
- Find connected components

Neo4J - Cypher (Query Language)

CREATE clause: creates nodes and edges

```
CREATE (n1:label {prop1:"val1", prop2:"val2"})
```

```
CREATE (n1)-[e1:label]->(n2)
```

```
CREATE (anna:Person {name:"Anna", age:27}),
       (peter:Person {name:"Peter", age:33}),
       (anna)-[e1:married {day:"2021-06-20"}]->(peter)
```

Example: Genealogy tree

```
CREATE
(anna:Person    {name:"Anna",gender:"f"}),   (peter:Person {name:"Peter", gender:"m"}),
(mike:Person    {name:"Mike",gender:"m"}),   (lilly:Person {name:"Lilly", gender:"f"}),
(max:Person     {name:"Max", gender:"m"}),   (susan:Person {name:"Susan", gender:"f"}),
(william:Person {name:"William", gender:"m"}),(john:Person {name:"John",  gender:"m"}),
(mary:Person    {name:"Mary", gender:"f"}),  (james:Person {name:"James", gender:"m"}),
(cora:Person    {name:"Cora", gender:"f"}), (george:Person {name:"George",gender:"m"}),
(carolyn:Person {name:"Carolyn", gender:"f"}),(richard:Person {name:"Richard", gender:"m"}),
(serena:Person  {name:"Serena", gender:"f"}),
 (anna)-[:child]->(max),     (anna)-[:child]->(susan),    (peter)-[:child]->(susan),
(peter)-[:child]->(william), (mike)-[:child]->(john),      (mike)-[:child]->(mary),
(lilly)-[:child]->(john),   (lilly)-[:child]->(mary),     (susan)-[:child]->(james),
(susan)-[:child]->(cora), (william)-[:child]->(george), (william)-[:child]->(carolyn),
 (john)-[:child]->(george),  (john)-[:child]->(carolyn),   (mary)-[:child]->(richard),
 (mary)-[:child]->(serena)
```

Neo4J - Cypher (Query Language)

`MATCH` clause: Pattern matching within the graph

```
(variable : node_label) -[variable : edge_label]-> ()
```

`WHERE`, `ORDER BY`, `LIMIT`, `RETURN`, `UPDATE`, ... clauses: as in SQL or XQuery

```
MATCH (p : person)-[:friend]-(x : person)
WHERE p.name = 'Peter'
ORDER BY x.name
RETURN x.name, x.city
```

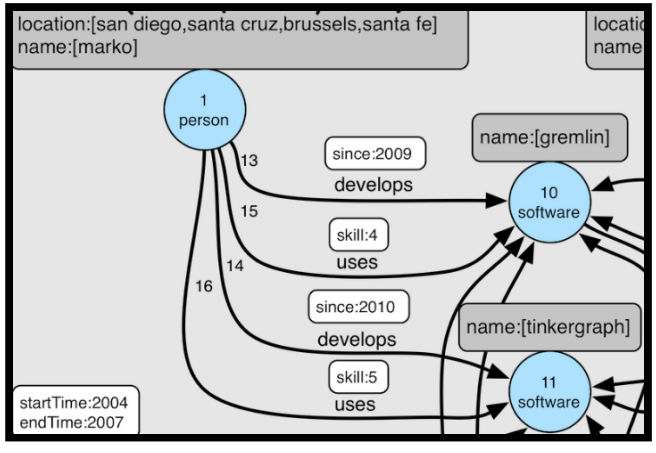
Within a MATCH clause `( )` represent a node, and `- -` / `- [ ] -` is an edge. Edges can have a direction `- ->` / `<- -`. If no direction is specified `- -`, both incoming and outgoing edges are matched. It is possible to introduce variables (here: p and x) to refer to a node or edge, e.g. within a `WHERE` predicate or the `RETURN` clause. After a colon, the node or edge label can be specified, e.g. `:person`. The query on this slide finds the names and cities of Peter's friends.

Gremlin (Graph Query Language)

```
gremlin> graph = TinkerFactory.createTheCrew()
gremlin> g = traversal().withEmbedded(graph)

// find Marko's person node
gremlin> g.V().has('person', 'name', 'marko')
==>v[1]

// which software did Marko develop?
gremlin> g.V().has('person', 'name', 'marko')
         .out('develops').hasLabel('software')
         .values('name')
==>gremlin
==>tinkergraph
```



Full graph at
https://bit.ly/thecrewgraph

Gremlin is a universal graph query language. Here, we use it to query an example graph within the embedded im-memory graph database Tinkergraph. The first two lines in the code creates a the example graph and initializes a traversable object `g`. With Gremlin, we traverse the graph by navigating over nodes and edges. In the longer example query on this slide, we first find all nodes `g.V()`, then we select only the person node with name Marko. After that we navigate to its neighbor nodes via outgoin `develops` edges, check wether their label is software, and finally return the value of the `name` property of that nodes.

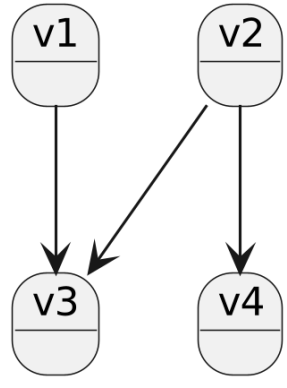
Graphs in relational databases

Vertex Table

vid
v1
v2
v3
v4

Edge Table

eid	source	target
e1	v1	v3
e2	v2	v3
e3	v2	v4

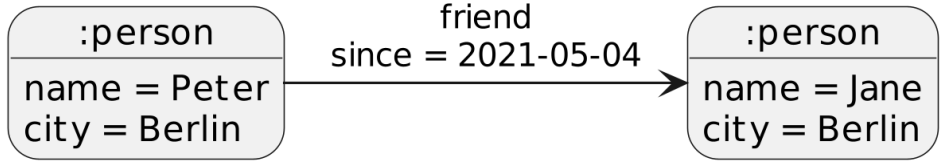


Which nodes have an outgoing edge to one of those nodes where v2 has an outgoing edge to?

```
SELECT DISTINCT vx.source FROM edges v2, vx
WHERE vx.target = v2.target AND v2.source = 'v2'
```

In a relational database, we can store vertices and edges in two separate tables. Instead of a unique edge id (here: eid), it is also possible to omit this column and use (soruce, target) as the primary key. Graph queries in SQL typically need a lot of joins between the vertex and edge tables.

Property graphs in rel. databases



People

pid	**name**	**city**
1	Peter	Berlin
2	Jane	Berlin

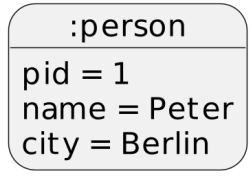
Friends

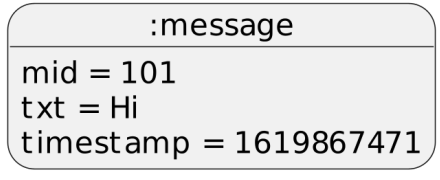
pid1	**pid2**	**properties**
1	2	{ "since": "2021-05-04" }

In a property graph, vertices and edges have a label and a set of properties. In this example, we create a table for each vertex label (people table), and a table for each edge label (friends). Other modelling alternatives can be found on the next slide. The people table is the vertex table plus two columns for two properties. friends is the edge table with a column of type JSON to store arbitrary edge properties.

Labels

Modelling alternatives for vertex / edge labels:





- One table for each label

  People (pid, name, city)
  Messages (mid, txt, timestamp)

- Label column

  Vertices (vid, label, name, city, txt, timestamp)

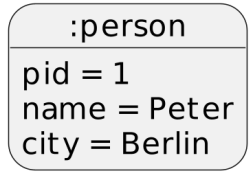
- Labels table

  Vertices (vid, name, city, txt, timestamp)
  Labels(vid, label)

The second and third approach is more flexible. New vertices with new labels can simply be inserted without creating a new table. But as these approaches lead to more complex and worse-understandable queries, the one-table-for-each-label approach is mostly used. The third approach allows vertices having multiple labels (e.g. person and student). The same three approaches can be used for edge labels, too.

Properties

Modelling alternatives for vertex / edge properties:

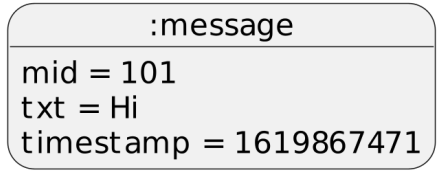


- One column for each property

  Vertices (vid, label, name, city, txt, timestamp)

- Properties column (JSON / XML / ...)

  Vertices (vid, properties)



- Properties table

  Vertices (vid)

  Properties(vid, property, value)

Combinations are possible.

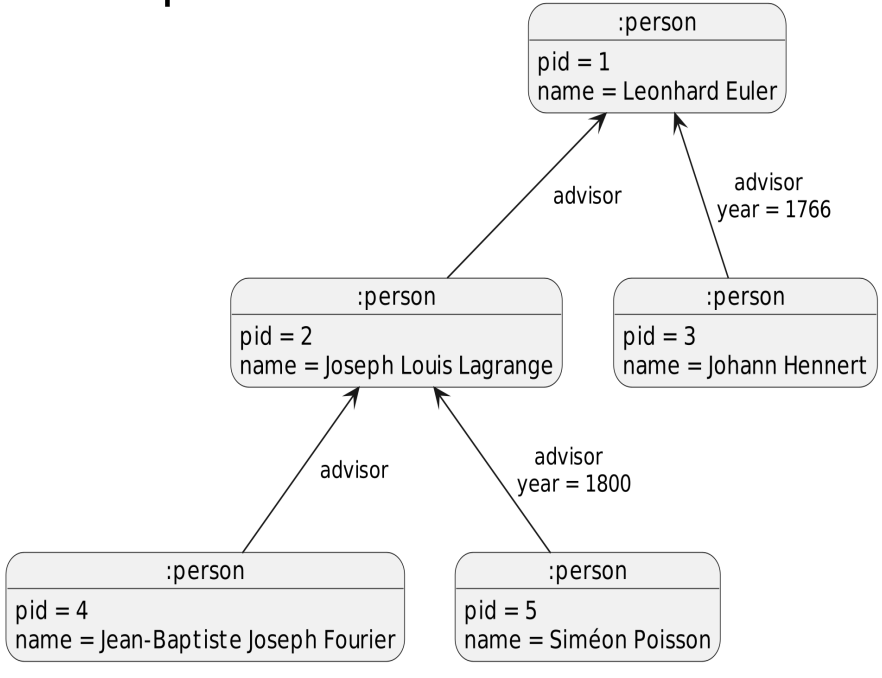
Alternative two and three is very flexible and allows using arbitrary properties. A problem of approach three is the data type of the value column. As different properties use different data types, a common type must be found, e.g. storing everything as a VARCHAR(2000000).
Another approach is combining two modelling alternatives, e.g. one and two: Important properties like name and timestamp become individual columns, all other properties are stored in a JSON column.

Tree structures in rel. databases

Tree: two vertices are connected by exactly one path.

Forest: two vertices are connected by at most one path.

pid	name	advisor	year
1	Leonard Euler	-	-
2	Joseph Louis Lagrange	1	-
3	Johann Hennert	1	1766
4	Jean-Baptiste Joseph Fourier	2	-
5	Siméon Poisson	2	1800



Sources: https://genealogy.math.ndsu.nodak.edu/ and https://academictree.org/

When a graph forms a tree, there is often no need for an edge table. In this example, every person has at most one doctor mother or doctor father. So we can directly store the pid of a person's advisor in the advisor column. The edge property "year" becomes another column.

Graph queries in SQL

How many students does every advisor have?

```
SELECT advisor.name, COUNT(student.pid) AS num_students
FROM phd advisor LEFT JOIN phd student
  ON advisor.pid = student.advisor
GROUP BY advisor.name;
```

name	num_students
Jean-Baptiste Joseph Fourier	0
Johann Hennert	0
Joseph Louis Lagrange	2
Leonard Euler	2
Siméon Poisson	0

Again, we use the phd table twice in the FROM clause to find all advisors of all students. Afterwards, we group by the advisor to count the number of their students.

CTE - Common Table Expressions

WITH ... AS (SELECT ...) SELECT ...

```
WITH advisors AS
(SELECT pid, advisor FROM phd WHERE advisor IS NOT NULL)
SELECT * FROM advisors;
```

pid	advisor
2	1
3	1
4	2
5	2

With classical SQL, we need to manually join each hop individually up to a desired depth. Without recursive SQL, it is not possible to compute the transitive closure (the advisor of the advisor of the advisor ...). Recursive queries is an extension of the CTE feature in SQL. The CTE on this slide simply find all pids and the pids of their advisor of those students who have a doctor mother or doctor father.

CTE - Common Table Expressions

```
WITH advisors AS (SELECT pid, advisor FROM phd
                                WHERE advisor IS NOT NULL),
advisors2 AS (SELECT c.pid, a.advisor FROM phd c JOIN phd a
                ON c.advisor = a.pid WHERE a.advisor IS NOT NULL)
SELECT * FROM advisors UNION ALL SELECT * FROM advisors2;
```

pid	advisor
2	1
3	1
4	2
5	2
4	1
5	1

As a next step, we introduce one additional hop. The first CTE finds all people and their direct advisor (see previous slide), and the second CTE `advisors2` finds a pid and the advisor of their advisor. In the result, we find two new rows: (4,1) and (5,1) indicating that person 1 has people 4 and 5 as an indirect student.

Recursive Queries

```
WITH RECURSIVE cte_name AS (
     SELECT initialvalues ....
       UNION [ALL]
     SELECT recursivevalues FROM ... cte_name ...
 )
   SELECT * FROM cte_name;
```

`SELECT initialvalues` must be a non-recursive expression that pre-populates the result table **cte_name** with some initial rows. For each of these rows, further rows can be added with the `UNION` clause and also for each recursively added row.

❗ **Caution:** Endless recursion has to be avoided

Recursive Queries (Examples)

Count from 1 to 1000:

```
WITH RECURSIVE count AS (
     SELECT 1 AS nr
       UNION
     SELECT nr+1 FROM count WHERE nr < 1000
 )
   SELECT * FROM count;
```

Calculate the ??? series:

```
WITH RECURSIVE x AS (
     SELECT 0 AS n1,1 AS n2
       UNION
     SELECT n2 AS n1,n1+n2 AS n2 FROM x WHERE n1 < 100000
 )
   SELECT n1 FROM x;
```

Recursive Queries

```
WITH RECURSIVE phd_closure AS (
 SELECT pid, advisor FROM phd WHERE advisor IS NOT NULL
 UNION ALL
 SELECT c.pid, a.advisor FROM phd_closure c
 JOIN phd a ON c.advisor = a.pid WHERE a.advisor IS NOT NULL
)
SELECT * FROM phd_closure;
```

pid	advisor

This query finds the transitive closure: all descendants of every advisor. For this example graph, the result is the same as on the previous slide because the hight of our tree is just 2. But in general, this query finds all direct and indirect advisors up to the roots of the forest.
`WITH RECURSIVE` indicates that we have a recursive SQL query. Within the CTE's FROM clause, we use the CTE itself. The CTE consists of two parts combined by `UNION ALL`. The part before the union, we compute the starting set. Here, we find all people who have an advisor. Let's call this set "phd_closure". After the union, we join phd_closure and the original phd table to find all advisors of those people in the set "phd_closure". As a result, they are added to the set and in the next iteration their advisors are found.
It is important that a recursive query must terminate. For our query, this is the case, because there are no cycles, and we stop for those people who's advisor is NULL. But even without this IS NOT NULL predicate, the recursion would stop, because these rows would not find a join partner in the next iteration.

Recursive Queries

```
WITH RECURSIVE phd_closure AS (
 SELECT pid, advisor, 1 AS level, ''||advisor as advisors
 FROM phd WHERE advisor IS NOT NULL
 UNION ALL
 SELECT c.pid, a.advisor, c.level+1, c.advisors||'->'||a.advisor
 FROM phd_closure c
 JOIN phd a ON c.advisor = a.pid WHERE a.advisor IS NOT NULL
)
SELECT * FROM phd_closure;
```

pid	advisor	level	advisors
2	1	1	1
3	1	1	1
4	2	1	2
5	2	1	2
4	1	2	2->1
5	1	2	2->1

Recursive Queries

```
WITH RECURSIVE phd_closure AS (
 SELECT pid, advisor FROM phd WHERE advisor IS NOT NULL
 UNION ALL
 SELECT c.pid, a.advisor FROM phd_closure c
 JOIN phd a ON c.advisor = a.pid WHERE a.advisor IS NOT NULL
)
SELECT advisor, count(*) FROM phd_closure GROUP BY advisor;
```

How many descendants (doctor children, grandchildren, grandgrandchildren, ...) does every advisor have?

advisor	count(*)
1	4
2	2

On the previous slides, we simply wrote `SELECT * FRO phd_closure` to output the full closure. Now we continue by using the closure as the input for a GROUP BY query. We compute the total number of direct and indirect students for each person.