Data Journalism

Big Data Engineering
(formerly Informationssysteme)

Prof. Dr. Jens Dittrich

bigdata.uni-saarland.de

July 6, 2023

Data Journalism

Planned structure for each ~~two~~ one-week lecture:

1. Concrete application: Data Journalism
2. What are the data management issues behind this?
3. Basics to be able to solve these problems
   (a) Slides
   (b) Jupyter/Python/SQL Hands-on
4. Transfer of the basics to the concrete application

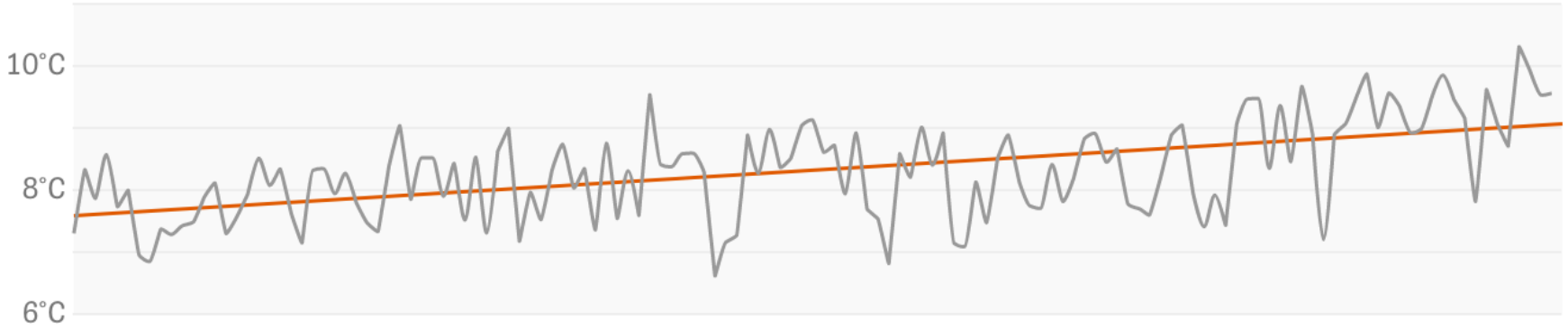
Data Journalism

1. Concrete application: Data Journalism

- Introduction
- Screenshots
- Examples

Es wird immer wärmer in Deutschland

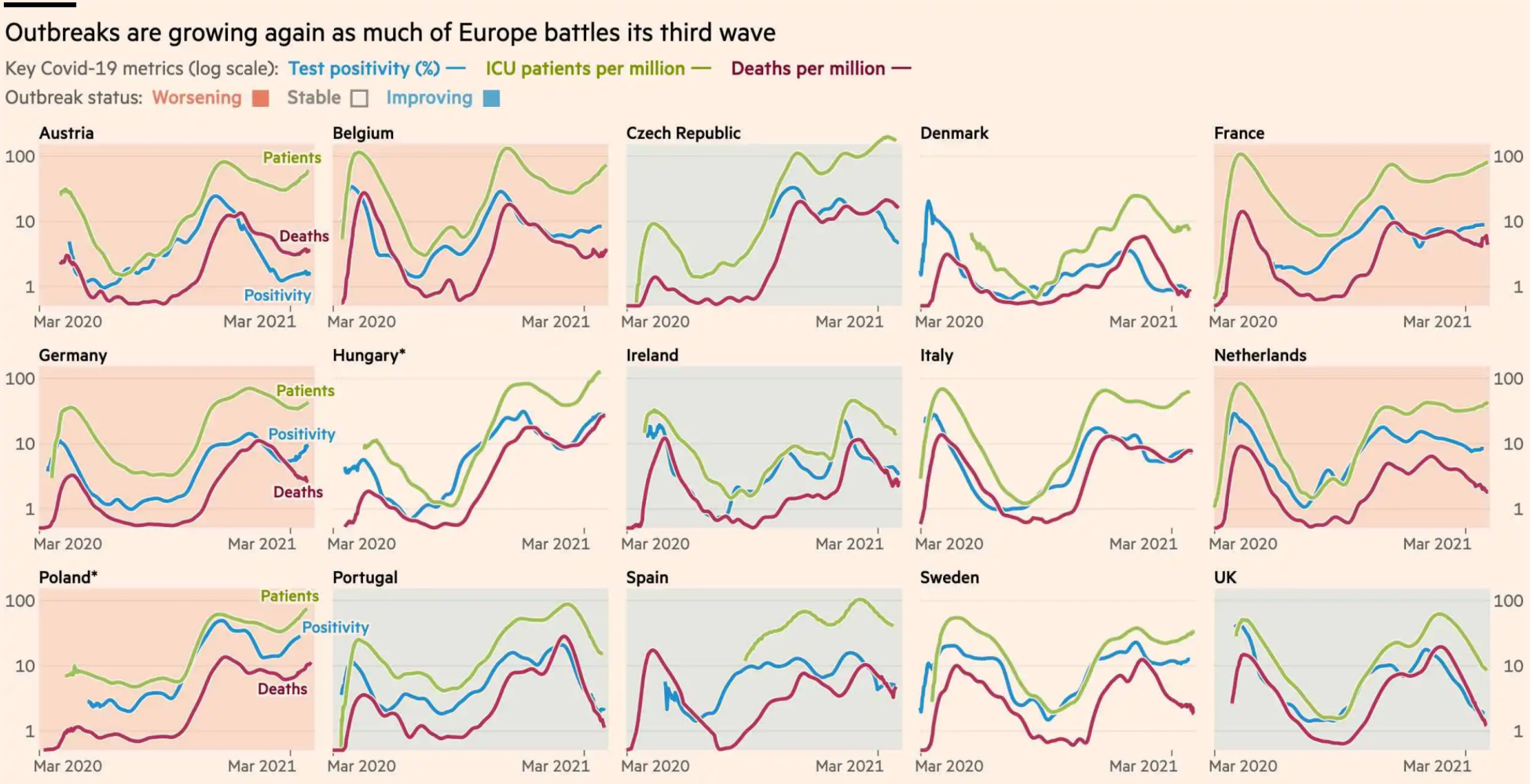
Langfristig gehen die Temperaturen im Jahresdurchschnitt seit Beginn der Wetteraufzeichnungen nach oben.



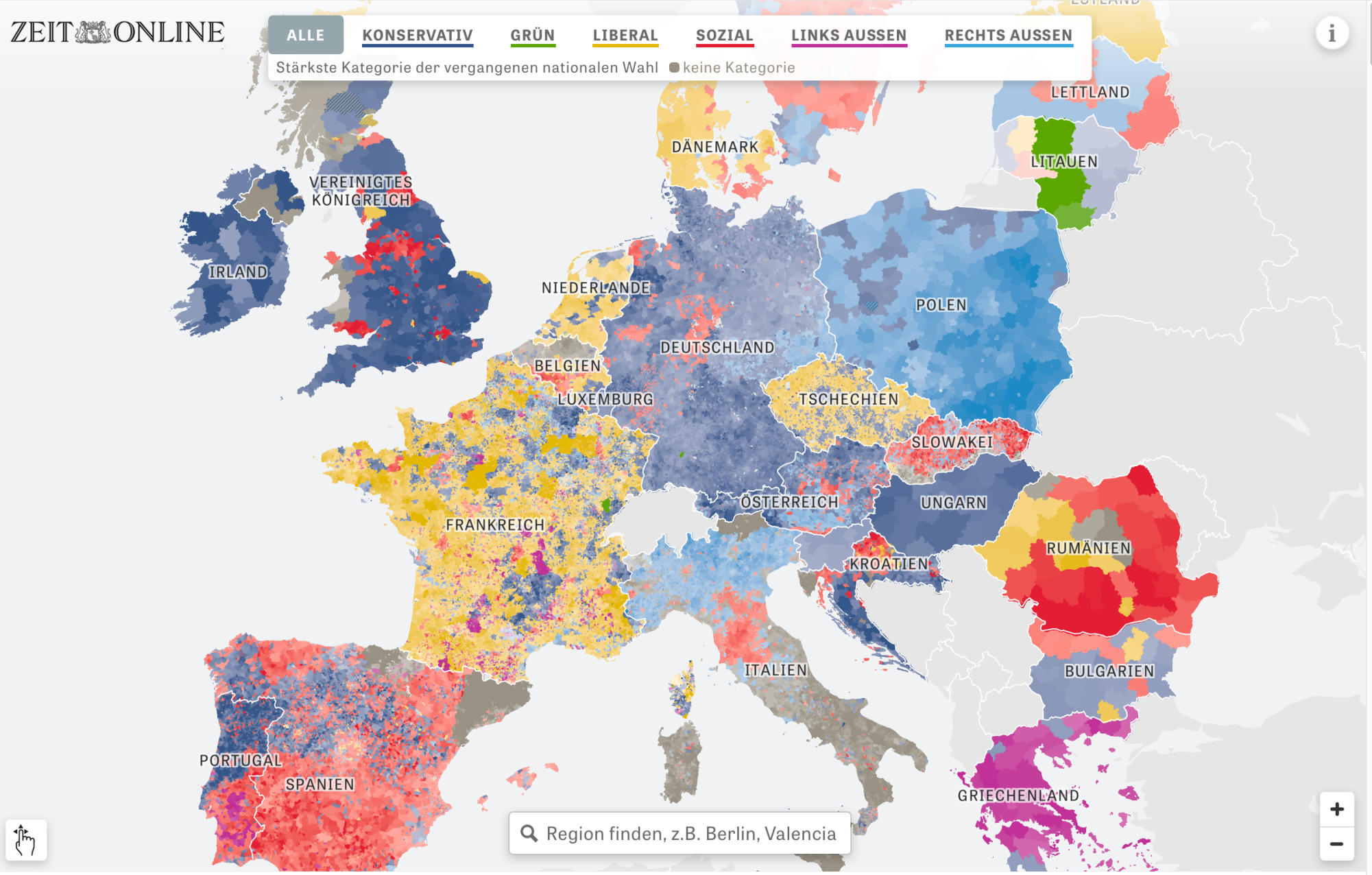
Quelle: Deutscher Wetterdienst, eigene Berechnungen

Im Mittel sind die Jahresdurchschnittstemperaturen hierzulande um 1,37 Grad Celsius angestiegen. Das zeigt die rote gerade Linie oben. Die Temperatur ist ein guter Indikator, weil sie in Messungen und Computermodellen gut und relativ genau zu handhaben ist.

https://www.zeit.de/wissen/umwelt/2018-08/wetter-hitze-juli-deutschland-rekord-sommer-klimawandel

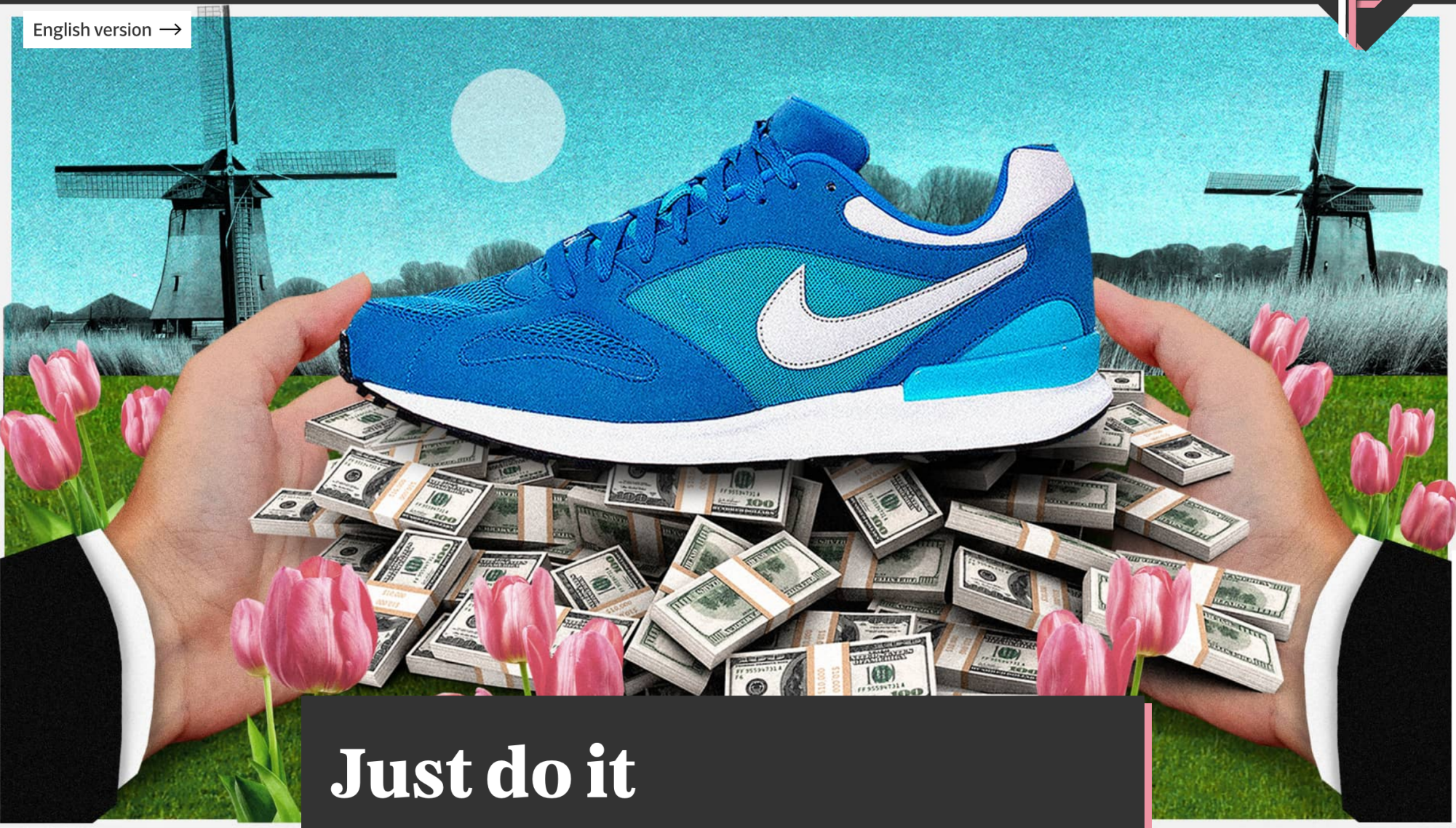


https://www.ft.com/content/d1af353b-709f-41d5-ac88-4ac38ee7dc6c



https://www.zeit.de/politik/ausland/2019-05/parlamentswahlen-eu-laender-wahlergebnisse-europakarte

English version →



Just do it

Nike ist bei sportlichen Wettkämpfen auf der ganzen Welt präsent. In einer Disziplin aber ist das Unternehmen selbst kaum zu schlagen - im Vermeiden

https://projekte.sueddeutsche.de/paradisepapers/wirtschaft/nike-und-die-niederlande-prellen-den-deutschen-staat-e116625/

Data Journalism

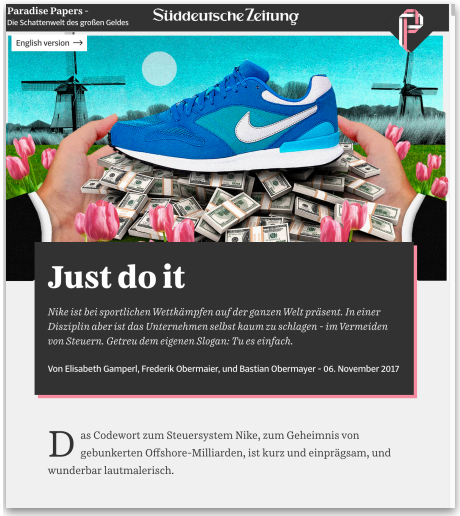
Data Journalism

Creation of articles with the help of data analysis tools or the (partial) integration of these tools into the article itself.

Incorporating data analysis into the text?

Wait, we've seen this before:

Newspaper article



Software Development

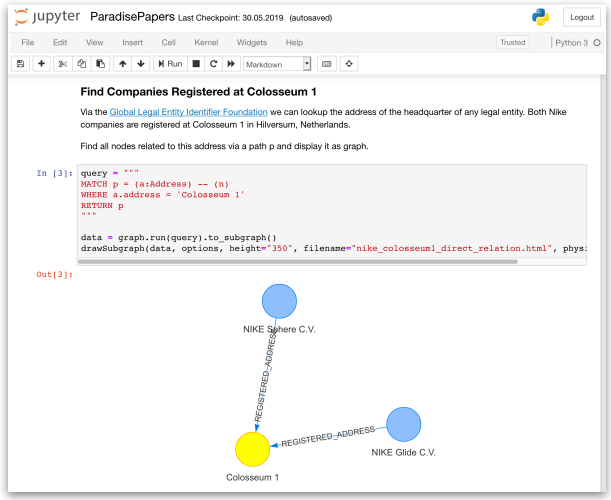
```
        return new_relation

class Equi_Join_HashBased(Equi_Join):
    def _dot(self, graph, prefix):
        return super()._dot(graph, prefix, "M_HashBased{},{}")

    def evaluate(self):
        l_eval_input = self.l_input.evaluate()
        r_eval_input = self.r_input.evaluate()

        # build new relation
        new_name = str(self)
        new_relation = Relation(new_name, self.get_schema())

        # compute attribute indexes
        l_idxs = [l_eval_input.get_attribute_index(attr) for attr in
        r_idxs = [r_eval_input.get_attribute_index(attr) for attr in

        # build hash table on left relation
        hash_table = dict()
        for tup in l_eval_input.tuples:
            join_pred = tuple([tup[i] for i in l_idxs])
            bucket = hash_table.get(join_pred, [])
            hash_table[join_pred] = bucket+[tup]

        # probe right relation against hash table
        result_tups = set()
        for tup_r in r_eval_input.tuples:
            join_pred = tuple([tup_r[i] for i in r_idxs])
            bucket = hash_table.get(join_pred, [])
            for tup_l in bucket:
                # add results tuples to relation
                new_relation.add_tuple(tup_l+tup_r)

        return new_relation

    def _dot(self, graph, prefix):
        # build name and label and call helper function
        node_name = prefix + 'Joi'
        node_label = "⋈{},{}".format(self.l_attrs, self.r_attrs)
        self._dot_helper(graph, prefix, node_name, node_label)
        return node_name


class Grouping_HashBased(Grouping):
    def _dot(self, graph, prefix):
        return super()._dot(graph, prefix, "γ_HashBased{} {}")
```

converges towards

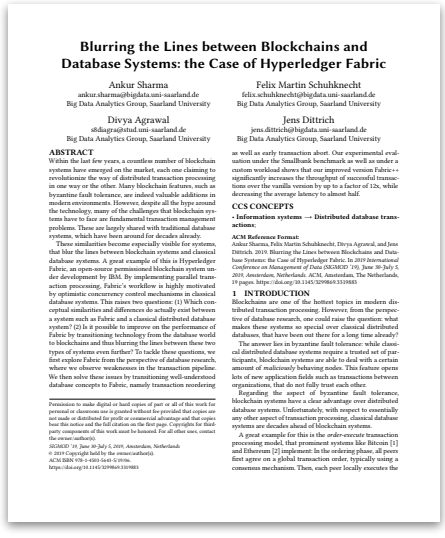
Hypertext+Code (e.g. Jupyter)

converges towards

converges towards



converges towards



Blurring the Lines between Blockchains and Database Systems: the Case of Hyperledger Fabric

Ankur Sharma
ankur.sharma@bigdata.uni-saarland.de
Big Data Analytics Group, Saarland University

Felix Martin Schuhknecht
felix.schuhknecht@bigdata.uni-saarland.de
Big Data Analytics Group, Saarland University

Divya Agrawal
s8diagra@stud.uni-saarland.de
Big Data Analytics Group, Saarland University

Jens Dittrich
jens.dittrich@bigdata.uni-saarland.de
Big Data Analytics Group, Saarland University

ABSTRACT

Within the last few years, a countless number of blockchain systems have emerged on the market, each one claiming to revolutionize the way of distributed transaction processing in one way or the other. Many blockchain features, such as byzantine fault tolerance, are indeed valuable additions in modern environments. However, despite all the hype around the technology, many of the challenges that blockchain systems have to face are fundamental transaction management problems. These are largely shared with traditional database systems, which have been around for decades already.

These similarities become especially visible for systems, that blur the lines between blockchain systems and classical database systems. A great example of this is Hyperledger Fabric, an open-source permissioned blockchain system under development by IBM. By implementing parallel transaction processing, Fabric's workflow is highly motivated by optimistic concurrency control mechanisms in classical database systems. This raises two questions: (1) Which conceptual similarities and differences do actually exist between a system such as Fabric and a classical distributed database system? (2) Is it possible to improve on the performance of Fabric by transitioning technology from the database world to blockchains and thus blurring the lines between these two types of systems even further? To tackle these questions, we first explore Fabric from the perspective of database research, where we observe weaknesses in the transaction pipeline. We then solve these issues by transitioning well-understood database concepts to Fabric, namely transaction reordering as well as early transaction abort. Our experimental evaluation under the Smallbank benchmark as well as under a custom workload shows that our improved version Fabric++ significantly increases the throughput of successful transactions over the vanilla version by up to a factor of 12x, while decreasing the average latency to almost half.

CCS CONCEPTS

• **Information systems** → **Distributed database transactions**;

ACM Reference Format:

Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. 2019. Blurring the Lines between Blockchains and Database Systems: the Case of Hyperledger Fabric. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, Amsterdam, The Netherlands, 19 pages. https://doi.org/10.1145/3299869.3319883

1 INTRODUCTION

Blockchains are one of the hottest topics in modern distributed transaction processing. However, from the perspective of database research, one could raise the question: what makes these systems so special over classical distributed databases, that have been out there for a long time already?

The answer lies in byzantine fault tolerance: while classical distributed database systems require a trusted set of participants, blockchain systems are able to deal with a certain amount of maliciously behaving nodes. This feature opens lots of new application fields such as transactions between organizations, that do not fully trust each other.

Regarding the aspect of byzantine fault tolerance, blockchain systems have a clear advantage over distributed database systems. Unfortunately, with respect to essentially any other aspect of transaction processing, classical database systems are decades ahead of blockchain systems.

A great example for this is the *order-execute* transaction processing model, that prominent systems like Bitcoin [1] and Ethereum [2] implement: In the ordering phase, all peers first agree on a global transaction order, typically using a consensus mechanism. Then, each peer locally executes the

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5643-5/19/06.
https://doi.org/10.1145/3299869.3319883

Research Paper

Data Journalism

2. What are the data management and analysis issues behind this?

Questions already clarified

How do we model the data? How do we query this data?

ER, relational model, relational algebra, SQL

But:

Question 1

How do we manage graphical data?

Question 2

How do we query this data?

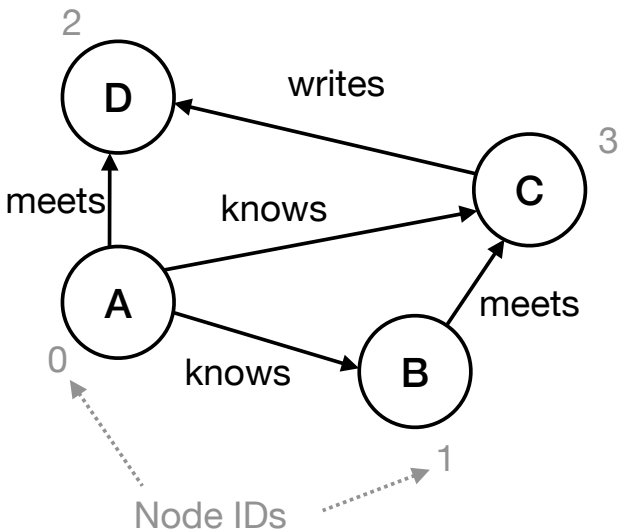
Data Journalism

3. Basics to be able to solve these problems
    (a) Slides
    (b) Jupyter/Python/SQL Hands-on

- Graphical data model: relational vs native model
- Queries on graphs: SQL vs Cypher

A Social Graph in the Relational Model

Example Graph:



Adjacency Matrix:

		to A_0	to B_1	to D_2	to C_3
from	A_0		knows	meets	knows
	B_1				meets
	D_2				
	C_3			writes	

- The adjacency matrix of the undirected graph already resembles a relation.
- However, this is a special case of a relation: a **pivot table**. Here, attributes are in the columns as well as in the rows.

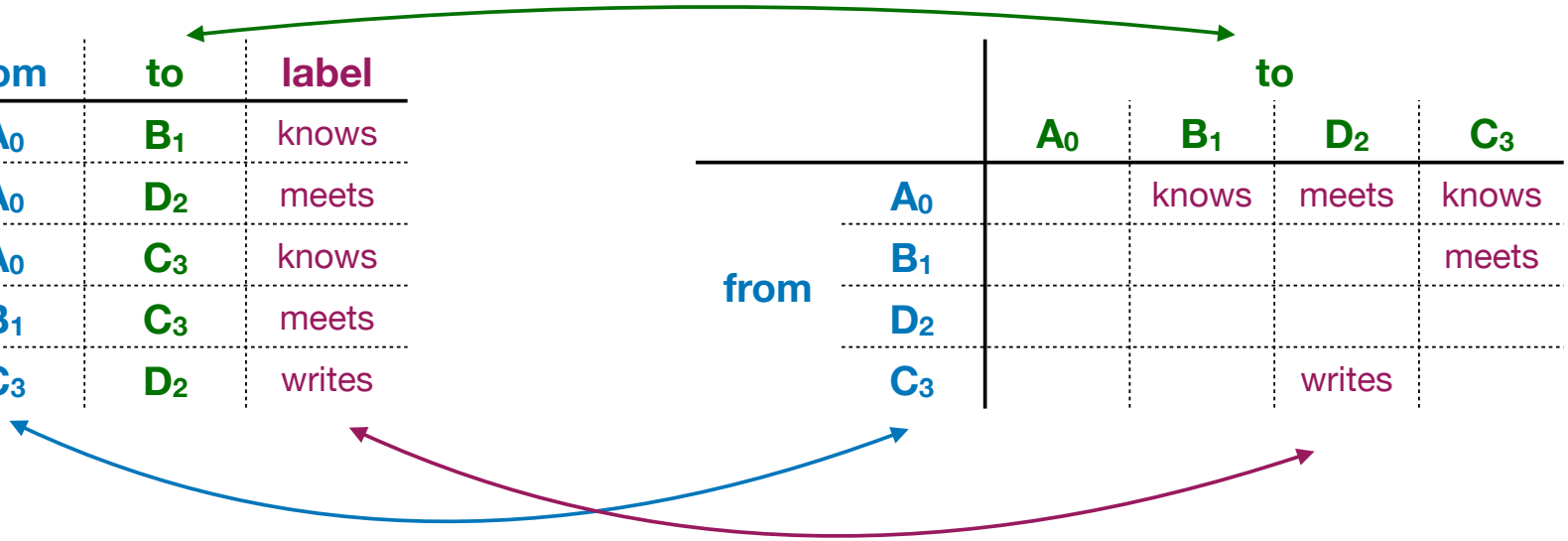
Pivot Tables

Adjacency Matrix (as Relation):

from	to	label
A_0	B_1	knows
A_0	D_2	meets
A_0	C_3	knows
B_1	C_3	meets
C_3	D_2	writes

Adjacency Matrix (as Pivot Table):

		to			
		A_0	B_1	D_2	C_3
from	A_0		knows	meets	knows
	B_1				meets
	D_2				
	C_3			writes	



Pivot Tables

In a pivot table, $x \geq 1$ attributes are displayed in the columns and $y \geq 1$ attributes are displayed in the rows (from an underlying relation). At the intersection of each column/row pair, $z \geq 1$ attributes of the underlying tuple of the relation are displayed.

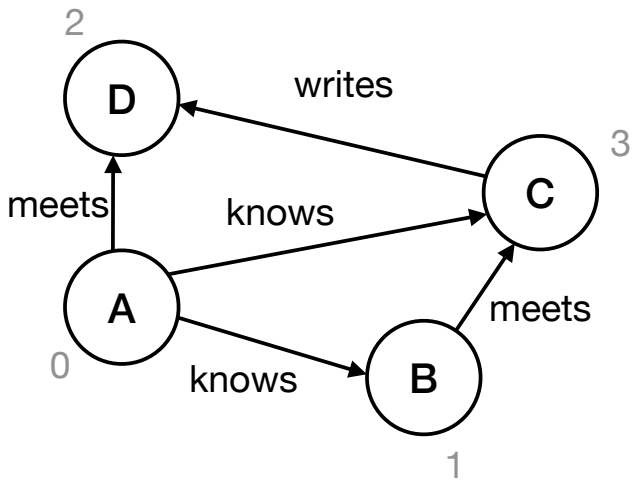
In SQL there is the command `PIVOT`, in PostgreSQL `\crosstabview`.

Pivot Tables: Notes

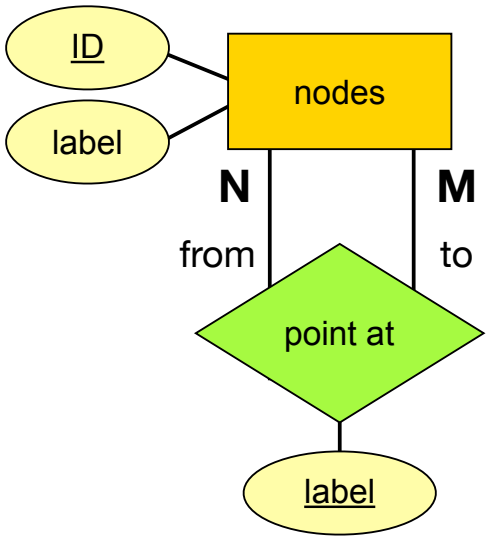
- if the underlying relation has not been grouped and aggregated across all attributes in x **and** y previously, multiple tuples can potentially appear in one cell
- that is why aggregation is usually done beforehand
- but this is not a must
- several tuples per cell can also be displayed or appropriately visualised

A Graph in the Relational Model: First Approach

Example Graph:



ER Model:



Relational Model:

[nodes] : {[ID:int, label:str]}
[point_at] : {[
from_ID:(nodes),
to_ID:(nodes),
label:str
]}

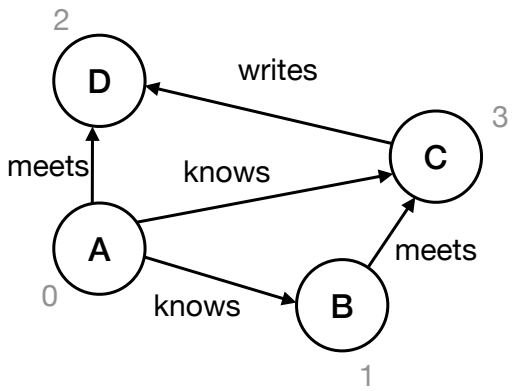
Sample data:

nodes	
ID	**label**
0	A
1	B
2	D
3	C

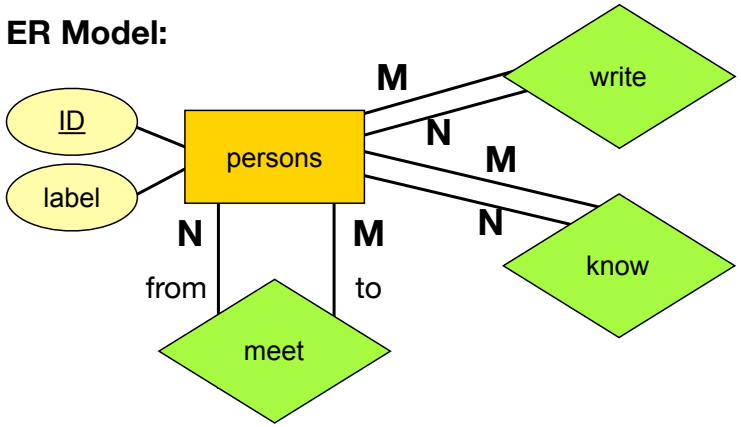
point_at		
from_ID	**to_ID**	**label**
0	2	meets
0	1	knows
0	3	knows
1	3	meets
3	2	writes

A Graph in the Relational Model: Second Approach

Example Graph:



ER Model:



Relational Model:

[persons] : {[ID:int, label:str]}
[meet] : {[
    from_ID:(persons),
    to_ID:(persons)
]}
[know] : {[
    from_ID:(persons),
    to_ID:(persons)
]}
etc.

Sample data:

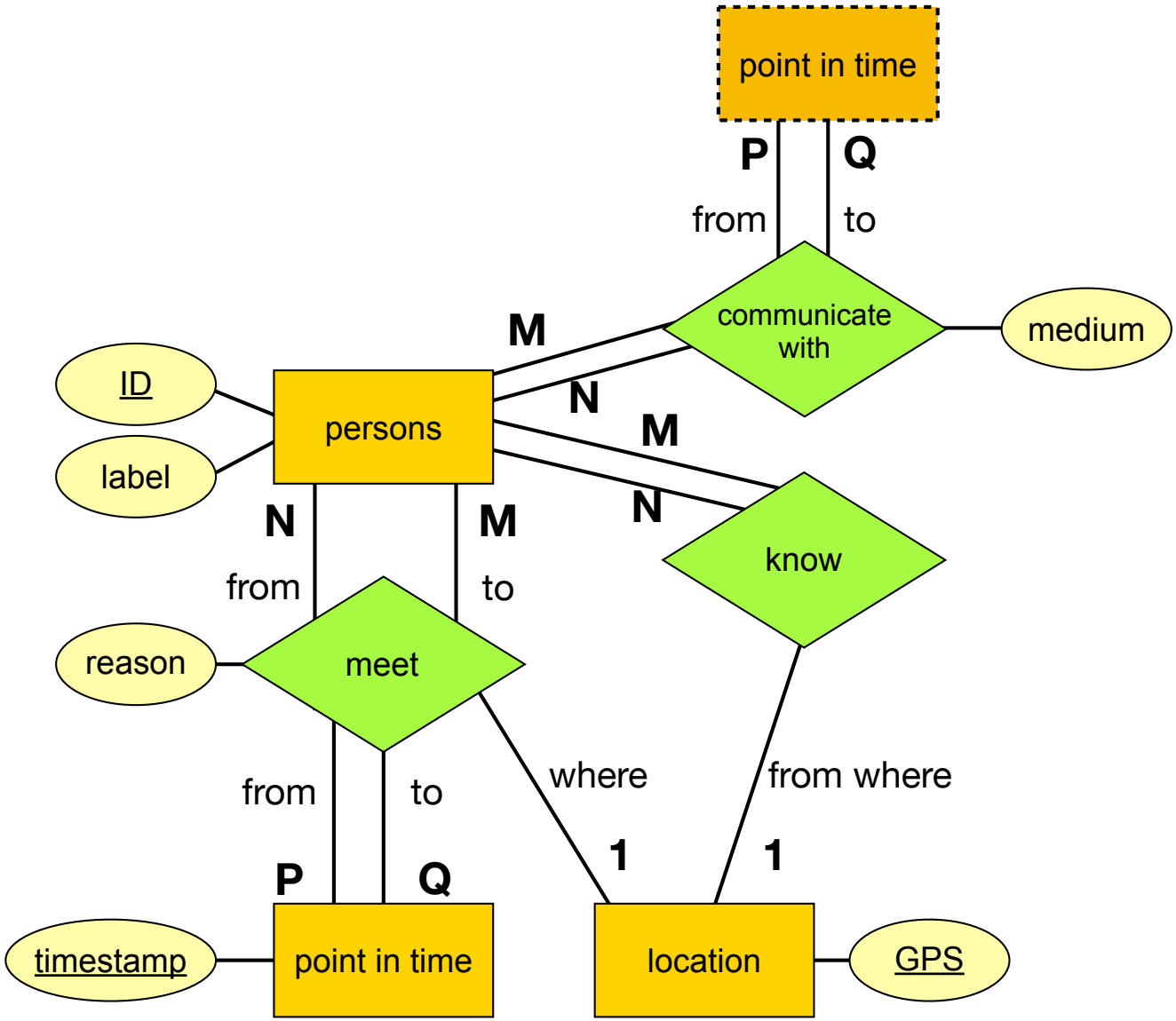
persons	
ID	**label**
0	A
1	B
2	D
3	C

meet	
from_ID	**to_ID**
0	2
1	3

know	
from_ID	**to_ID**
0	1
0	3

write	
from_ID	**to_ID**
3	2

A Graph in the Relational Model: Third Approach



Interim Conclusion: Graphs in the Relational Model

Interim Conclusion

The relational model is perfectly suited for modelling/storing graphs. We can model graphs in ER or in the relational model in quite different ways. A new/separate data model is not necessary **for conceptual reasons**.

Interim Conclusion:

- Graphical data model: native vs relational model

and the winner is:

the Relational Model

What about queries?

We'll have to dig deeper ...

SQL: Local Views With `WITH` (Common Table Expressions)

We had views in SQL with `CREATE VIEW` in the SQL-Notebook.

Local Views With `WITH` (Common Table Expressions, CTEs)

`WITH` defines a local view that is only visible within a specific SQL query.
In contrast to `CREATE VIEW`, this local view is not stored in the system.

Global View:

After defining this view, it can be used as often as desired.

```
CREATE VIEW foo AS (
     SELECT *
     FROM A JOIN B
          ON A.id = B.id
);

SELECT * FROM foo;
```

Yes, these are two statements.

vs

Local View:

After definition only usable within this query.

```
WITH foo AS (
     SELECT *
     FROM A JOIN B
          ON A.id = B.id
)
SELECT * FROM foo;
```

Yes, this is just one statement.

More About the `WITH` Syntax

The columns of a local view can be named:

```
WITH foo(n, g) AS (
     VALUES (1, 42)
)
SELECT * FROM foo;
```

is equivalent to:

```
WITH foo AS (
     SELECT 1 AS n, 42 AS g
)
SELECT * FROM foo;
```

In the following example, the attributes of the output relation have not been named:

```
WITH foo AS (
     SELECT 1 , 42
)
SELECT * FROM foo;
```

Output in PostgreSQL:

	?column? integer	?column? integer
1	1	42

Output in DuckDB:

	1	42
0	1	42

WITH RECURSIVE Semantics

Principle:

```
WITH RECURSIVE foo AS (
      nonRecursiveTerm()
  UNION
      recursiveTerm(foo)
)
SELECT * FROM foo;
```

Algorithm:

tmp = nonRecursiveTerm()
foo = tmp
As long as tmp $\neq \{\}$:
    tmp = recursiveTerm(tmp)
    foo $\cup=$ tmp
return foo

- UNION: Set semantics for the entire algorithm
- UNION ALL: **Multi**set semantics for the entire algorithm
- see DuckDB documentation on WITH/Common Table Expressions
- see PostgreSQL documentation on WITH/Common Table Expressions

WITH RECURSIVE Example with UNION

Example Query:

```
WITH RECURSIVE foo(n) AS (
      VALUES (1)
  UNION
      SELECT n+1 FROM foo
      WHERE n < 3
)
SELECT * FROM foo;
```

Algorithm:

tmp = nonRecursiveTerm()
foo = tmp
As long as tmp ≠ {}:
    tmp = recursiveTerm(tmp)
    foo ∪= tmp
return foo

Algorithm (Trace):

tmp = {(1)}
foo = tmp

% 1. Iteration:
tmp = recursiveTerm({(1)})
% tmp = {(2)}
foo ∪= tmp
% foo = {(1), (2)}

% 2. Iteration:
tmp = recursiveTerm({(2)})
% tmp = {(3)}
foo ∪= tmp
% foo = {(1), (2), (3)}

% 3. Iteration:
tmp = recursiveTerm({(3)})
% tmp = {()}
foo ∪= tmp
% foo = {(1), (2), (3)}

% As long as: tmp ≠ {}: Terminating the loop

return {(1), (2), (3)}

WITH RECURSIVE Example with UNION ALL

Example Query:

```
WITH RECURSIVE foo(n) AS (
      VALUES (1),(2),(3)
  UNION ALL
      SELECT n+1 FROM foo
      WHERE n < 3
)
SELECT * FROM foo;
```

Algorithm:

tmp = nonRecursiveTerm()
foo = tmp
As long as tmp $\neq \{\}$:
    tmp = recursiveTerm(tmp)
    foo $\cup$= tmp
return foo

Algorithm (Trace):

tmp = [(1), (2), (3)]
foo = tmp

% 1. Iteration:
tmp = recursiveTerm([(1), (2), (3)])
% tmp = [(2), (3)]
foo $\cup$= tmp
% foo = [(1), (2), (3), (2), (3)]

% 2. Iteration:
tmp = recursiveTerm([(2), (3)])
% tmp = [(3)]
foo $\cup$= tmp
% foo = [(1), (2), (3), (2), (3), (3)]

% 3. Iteration:
tmp = recursiveTerm([(3)])
% tmp = []
foo $\cup$= tmp
% foo = [(1), (2), (3), (2), (3), (3)]

% As long as: tmp $\neq \{\}$: Terminating the loop

return [(1), (2), (3), (2), (3), (3)]

Graphs with SQL (Graphs in SQL.ipynb)

Recursive Queries

The following query demonstrates the use of recursive queries and outputs the sum of all integers between 1 and 100. It first defines a recursive subquery `t` with the argument `n` which can then be reused in the body of the query (in parentheses following the `AS`). This body first defines a base case for the recursion, setting the argument `n` to the constant value 1. It then appends `n+1` to `t` using `UNION ALL` in each recursive step as long as `n` is smaller than 100. The outer query at the end then simply outputs the sum of all `n` contained in `t`.

In [10]:
```
cur = conn.cursor()
cur.execute("""
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
""")
print_set_postgres(cur)
cur.close()
```

```
[Result] : {[ sum ]}
{
        (5050)
}
```

see: Graphs in SQL.ipynb

Interim Conclusion: SQL as a Query Language for Graphs

Interim Conclusion

SQL is quite suitable for very simple queries on graphs; for more complex queries, however, only to a limited extent. Recursive queries in SQL quickly become difficult to read (and thus difficult to maintain).

What are the alternatives?

Cypher and Neo4j

Cypher

Cypher is one (of many) examples of a **domain-specific query language**. Cypher is specialised in graphs and is currently supported by several systems.

Cypher:

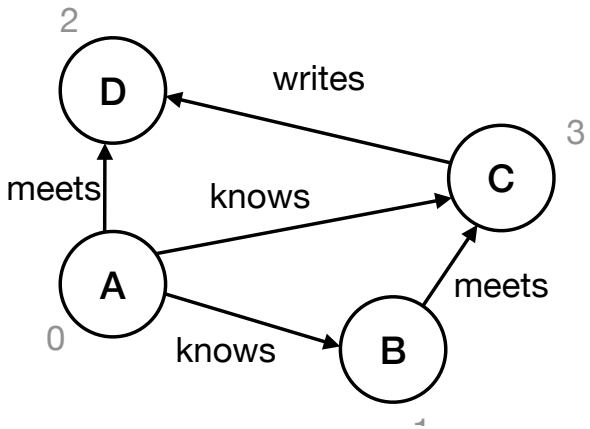
- Developed since 2011
- Open Cypher (since 2015): https://www.opencypher.org
- Cypher Tutorial: https://neo4j.com/developer/cypher-query-language/
- Details on the language definition: Cypher: An Evolving Query Language for Property Graphs, SIGMOD 2018

Neo4j:

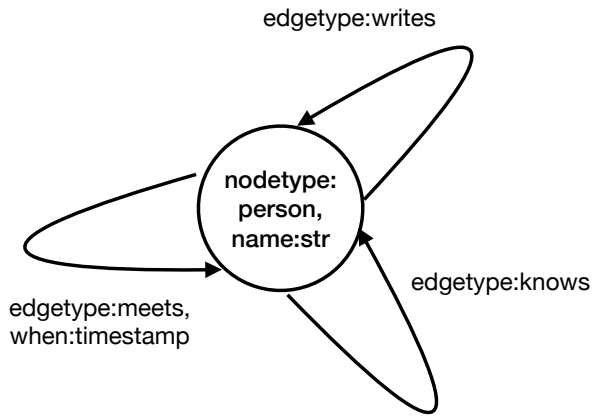
- open source and commercial licence
- https://github.com/neo4j
- https://neo4j.com/

Graph Schema in Neo4j?

Example Graph (Instance):



Example Graph ("Graph Schema"):



(Almost) schema-free Graphs

Unfortunately, a real schema like in SQL cannot be defined in Neo4j.

Why "almost"? What we can constrain is:

- nodes and edges are typed
- a certain attribute must exist (but other attributes can be added as desired)
- keys via attributes (similar to SQL)
- see Neo4j documentation: Constraints (some only available in the Enterprise Edition...)

Basic Structure of Cypher Queries: MATCH and RETURN

In the following, there are nodes of the type PERSON and edges of the types WRITES, MEETS, KNOWS.

MATCH and RETURN

```
MATCH (n:PERSON)
RETURN n.node_id AS node_id, n.name AS name;
```

Returns all nodes of type PERSON and displays their attributes node_id and name.

Result:

```
(0, A),
(1, B),
(3, C),
(2, D)
```

- MATCH (n:PERSON) corresponds to FROM person AS n in SQL
- RETURN corresponds to SELECT in SQL
- () is the so-called ASCII query syntax for a node

ASCII Query Syntax: ()-[]->()

```
MATCH ()-[r:KNOWS]->() RETURN r;
```

Returns all edges of the type KNOWS. The direction of the edge is taken into account.

Result:

```
((A)-[:KNOWS {}]->(C)),
((A)-[:KNOWS {}]->(B))
```

```
MATCH ()-[r:KNOWS]-() RETURN r;
```

Returns all edges of the type KNOWS. The direction of the edge is **not** taken into account here. Thus, each matching edge is returned twice.

Result:

```
((A)-[:KNOWS {}]->(C)),
((A)-[:KNOWS {}]->(B)),
((A)-[:KNOWS {}]->(B)),
((A)-[:KNOWS {}]->(C))
```

WITH

WITH in Cypher is very similar to WITH in SQL and also defines a local view in Cypher.

```
MATCH cyclePath = (t:TRANSACTION) - [*..6] -> (t)
WITH NODES(cyclePath) as path
RETURN path[0].node_id AS startID;
```

Returns all cycles up to maximum length 6.

- NODES() extracts the contained nodes as a list from each path selected with MATCH
- this list can be referenced by index: path[0]

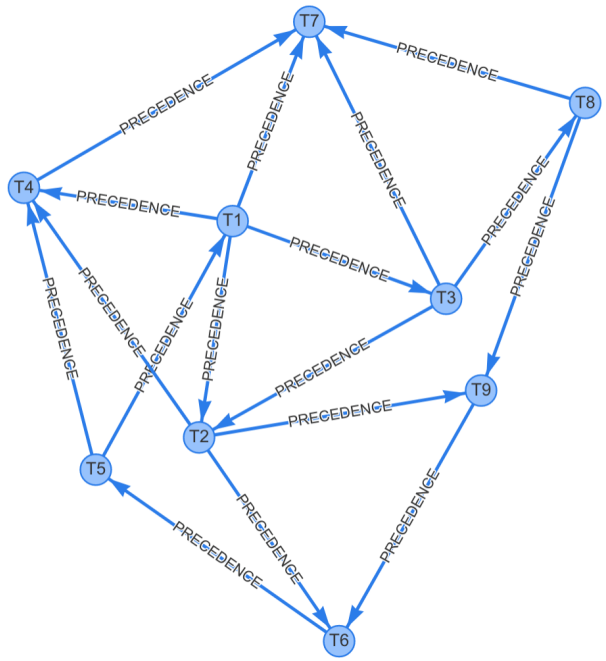
Graphs with Cypher (Graphs in Cypher.ipynb)

Advanced Precedence Graph

In the following, we will extend the basic example from the lecture slides to a total of nine transactions containing multiple possible cycles.

In [22]:
```
data_neo4j = graph.run("""
MATCH edges = (a:TRANSACTION) --> (b:TRANSACTION)
RETURN edges;
""").to_subgraph()
drawSubgraph(data_neo4j, options, height="500", filename="advanced_example.html", physics=physics, node_shape=node_shap
```

Out[22]:



Graphs in Cypher.ipynb

Interim Conclusion

Interim Conclusion: Queries on Graphs in SQL vs Cypher

For queries on graphs, Cypher is a lean and elegant language and much better suited than SQL.

Interim Conclusion:

- Cypher vs SQL

and the winner is:

Cypher

Overall Result

- Graphical data model: native vs relational model

and the winner is:

the Relational Model

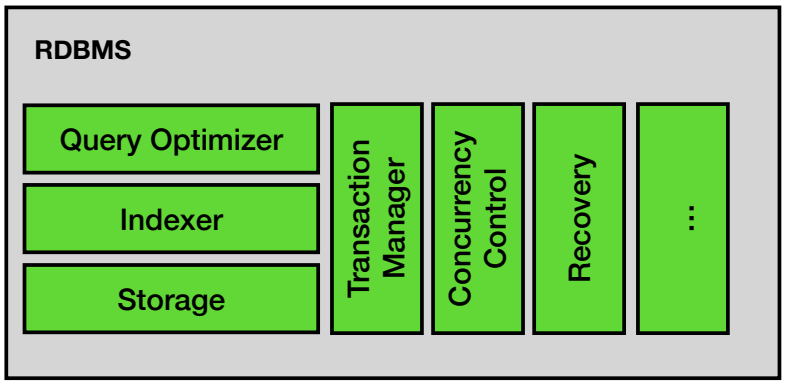
- Cypher vs SQL

and the winner is:

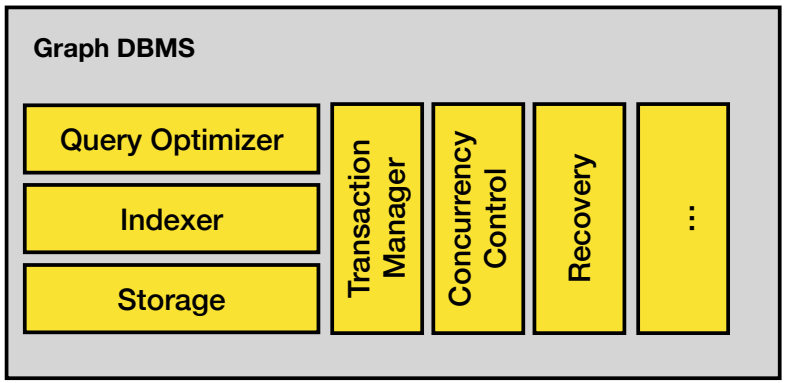
Cypher

RDBMS vs GraphDBMS vs Extended RDBMS

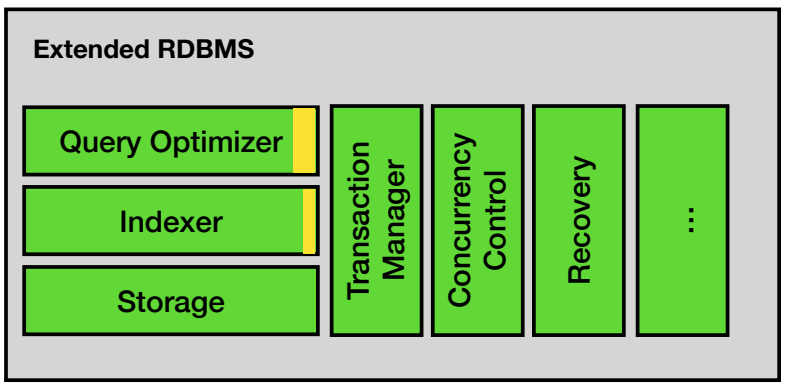
RDBMS, no or cumbersome support for graphs in SQL, possibly too slow for large graphs
Example: PostgreSQL



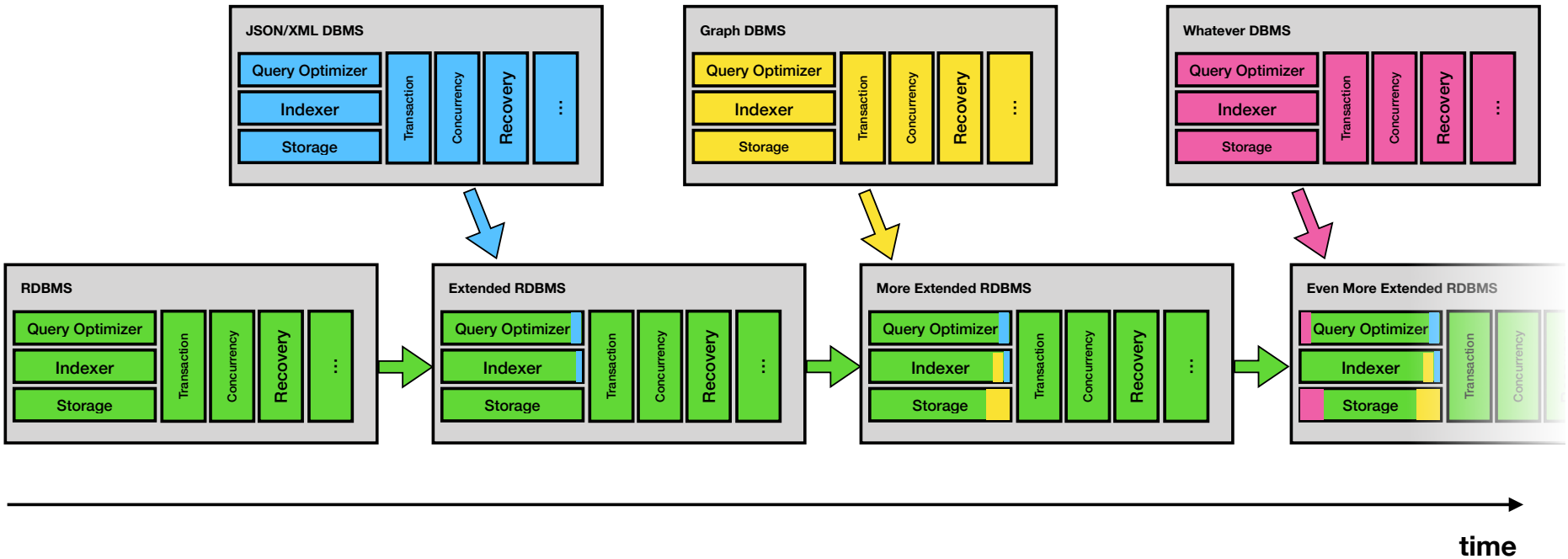
GraphDBMS, no SQL, unsuitable for more complex relational queries
Example: Neo4j



extended DBMS, the best of both worlds
Example: PostgreSQL+AGE

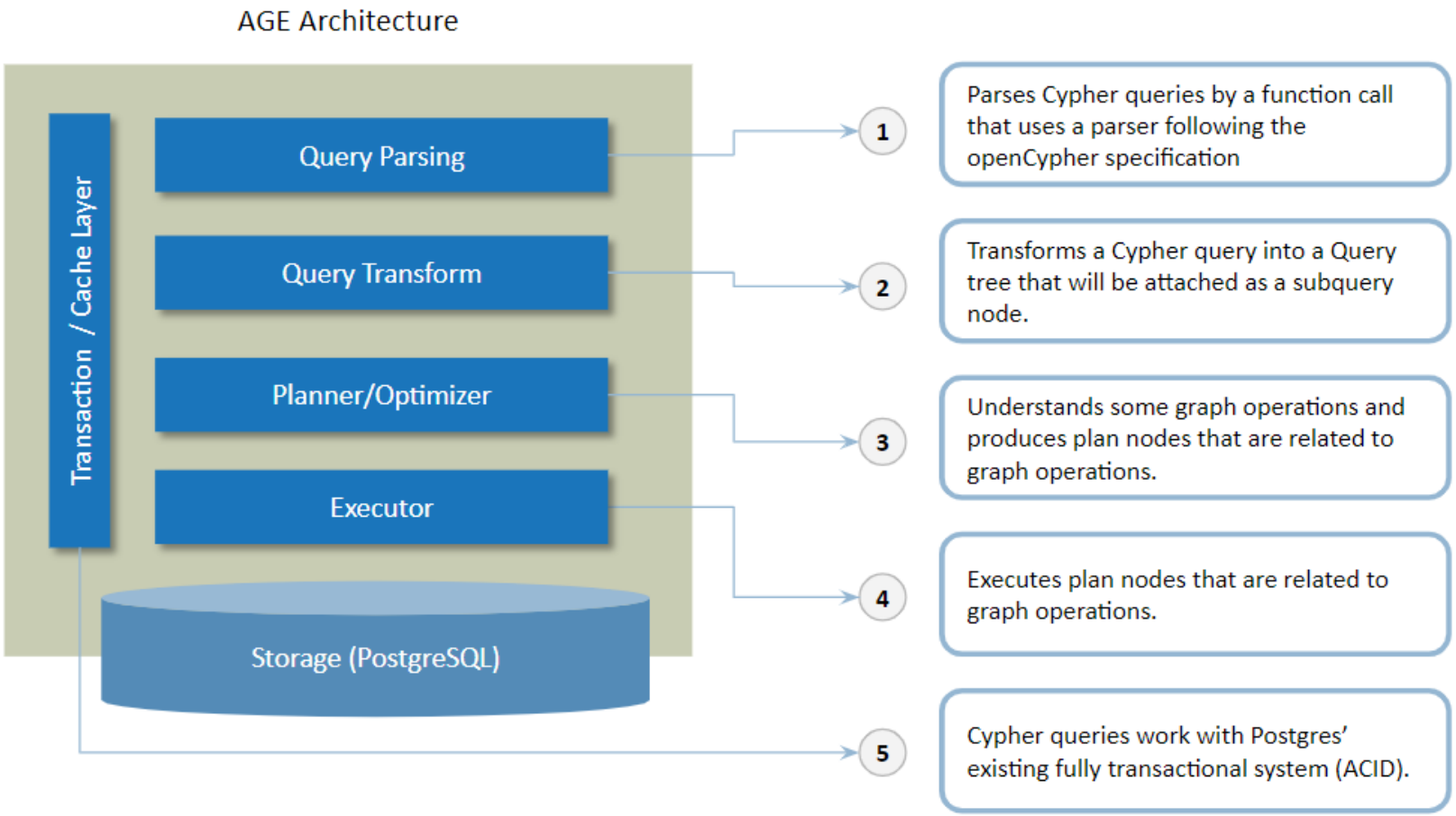


Development of Specialised DBMSs Over Time



- (claimed) advantages through specialised DBMSs are usually only relevant for a transitional/short period of time
- RDBMS vendors each extend their systems so that the advantages of specialised DBMSs fade over time
- however, not all extensions from research always make it into freely available DBMSs

Example for Graphs: AGE (A Graph Extension)



- Source: `https://age.apache.org/?l=overview`
- has recently become a top-level Apache project: heise

Integration of Cypher in SQL

Integration by means of a function call:

```
SELECT * FROM cypher('graph_name', $$
   MATCH (v)
   RETURN v
$$) as (v agtype);
```

The function cypher() generates a table as return value and can be integrated into the existing query optimiser very easily as a sub-query.

Integration of SQL in Cypher in SQL

Define a function (aka user-defined function, UDF) in SQL:

Create Function

```
CREATE OR REPLACE FUNCTION public.get_event_year(name agtype) RETURNS agtype AS $$
        SELECT year::agtype
        FROM history AS h
        WHERE h.event_name = name::text
        LIMIT 1;
$$ LANGUAGE sql;
```

Query

```
SELECT * FROM cypher('graph_name', $$
        MATCH (e:event)
        WHERE e.year < public.get_event_year(e.name)
        RETURN n.name
$$) as (n agtype);
```

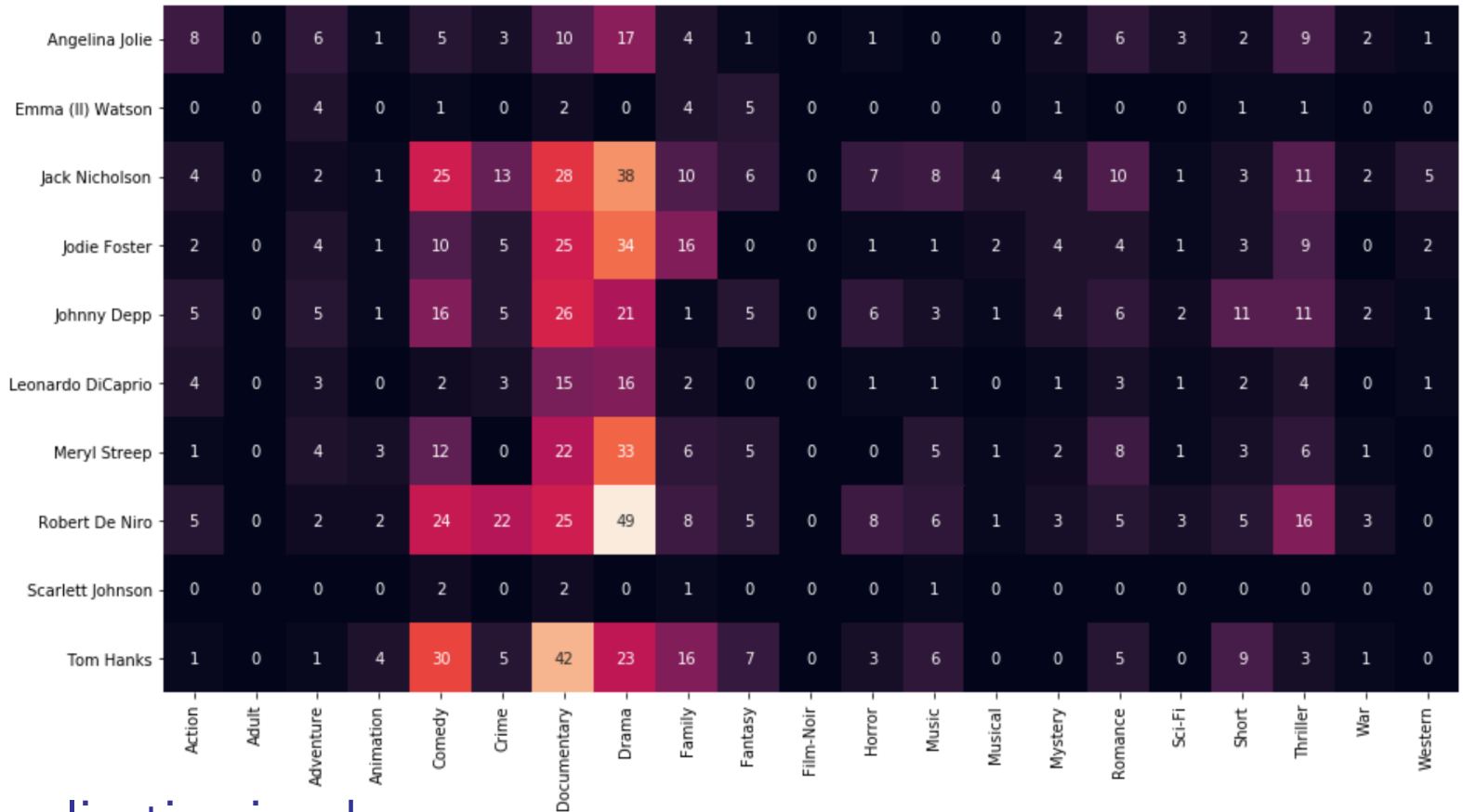
Scalar functions only, set-returning functions are currently not supported

Source

Data Visualisation in Jupyter (Data Visualization.ipynb)

```
# Plot heatmap
sns.heatmap(pivottable, ax=ax[0], annot=True,
            xticklabels=genres, yticklabels=actors, cbar=False)#, cmap="YlGnBu", cbar=False)

# Plot heatmap with other color scheme
sns.heatmap(pivottable, ax=ax[1], annot=True,
            xticklabels=genres, yticklabels=actors, cbar=False, cmap="YlGnBu")
```

Out[8]: <matplotlib.axes._subplots.AxesSubplot at 0x124b5bd68>



Data Visualization.ipynb

Summary

2. What are the data management and analysis issues behind this?

Question 1

How do we manage graphical data?

relational model

Question 2

How do we query this data?

Cypher (preferably fully integrated into a relational DBMS)