Datenjournalismus (Teil 1)

VL Big Data Engineering
(aka Informationssysteme)

Prof. Dr. Jens Dittrich

bigdata.uni-saarland.de

9. Oktober 2020

Datenjournalismus

Geplante Struktur für jeweils zwei Wochen Vorlesung:

1. Konkrete Anwendung: Datenjournalismus
2. Was sind die Datenmanagement und -analyseprobleme dahinter?
3. Grundlagen, um diese Probleme lösen zu können
   (a) Folien
   (b) Jupyter/Python/SQL Hands-on
4. Transfer der Grundlagen auf die konkrete Anwendung

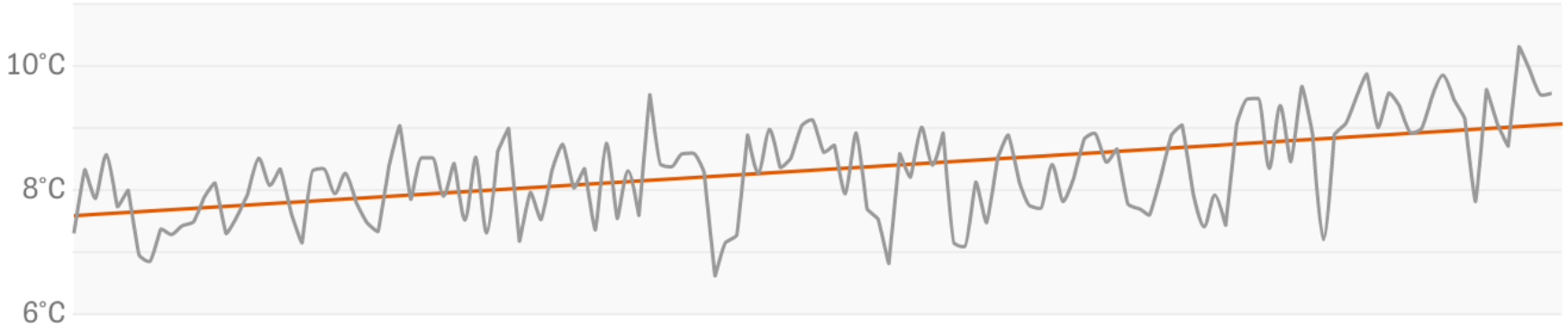
Datenjournalismus

1. Konkrete Anwendung: Datenjournalismus

- Einführung
- Screenshots
- Beispiele

Es wird immer wärmer in Deutschland

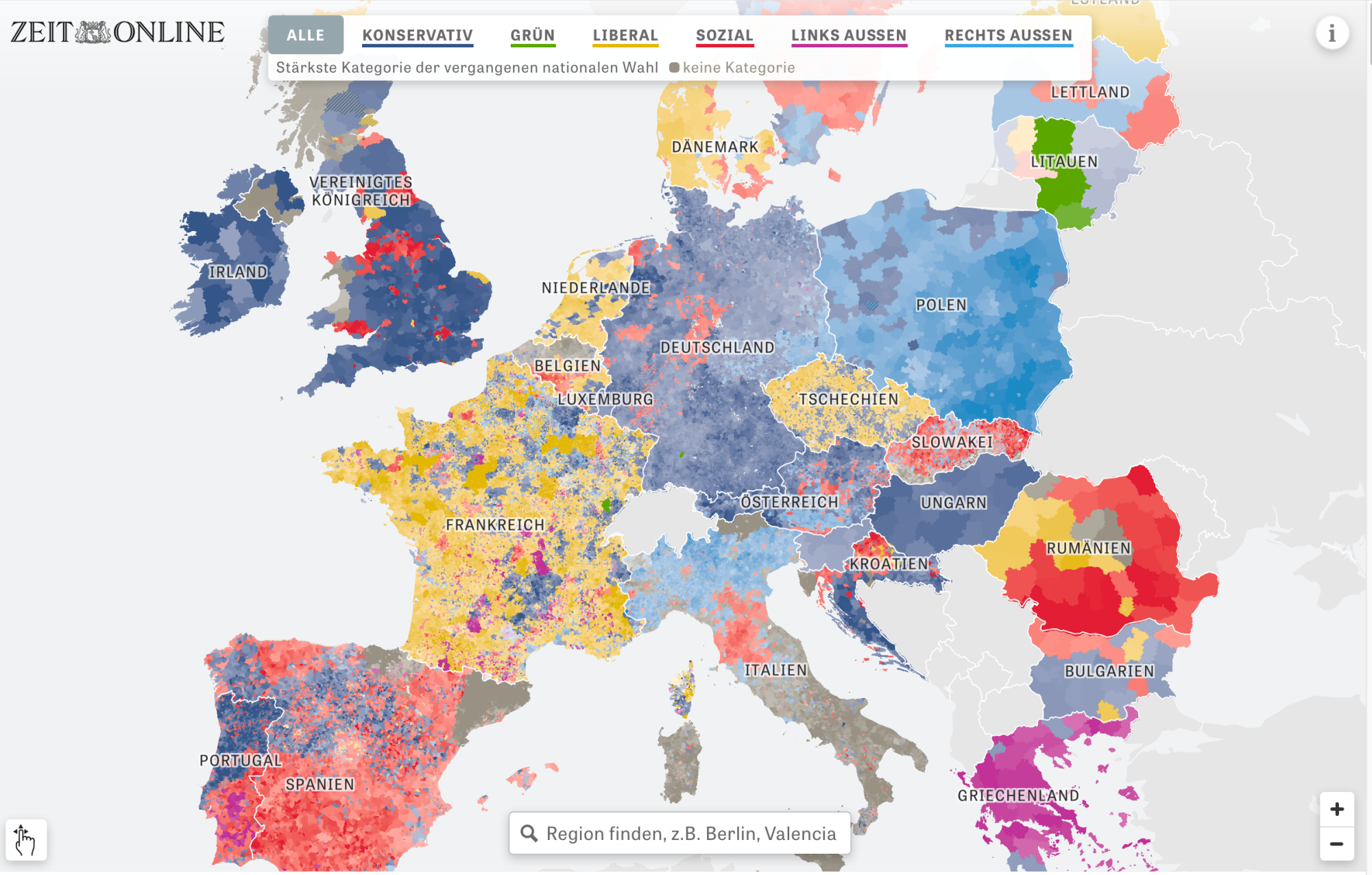
Langfristig gehen die Temperaturen im Jahresdurchschnitt seit Beginn der Wetteraufzeichnungen nach oben.



Quelle: Deutscher Wetterdienst, eigene Berechnungen

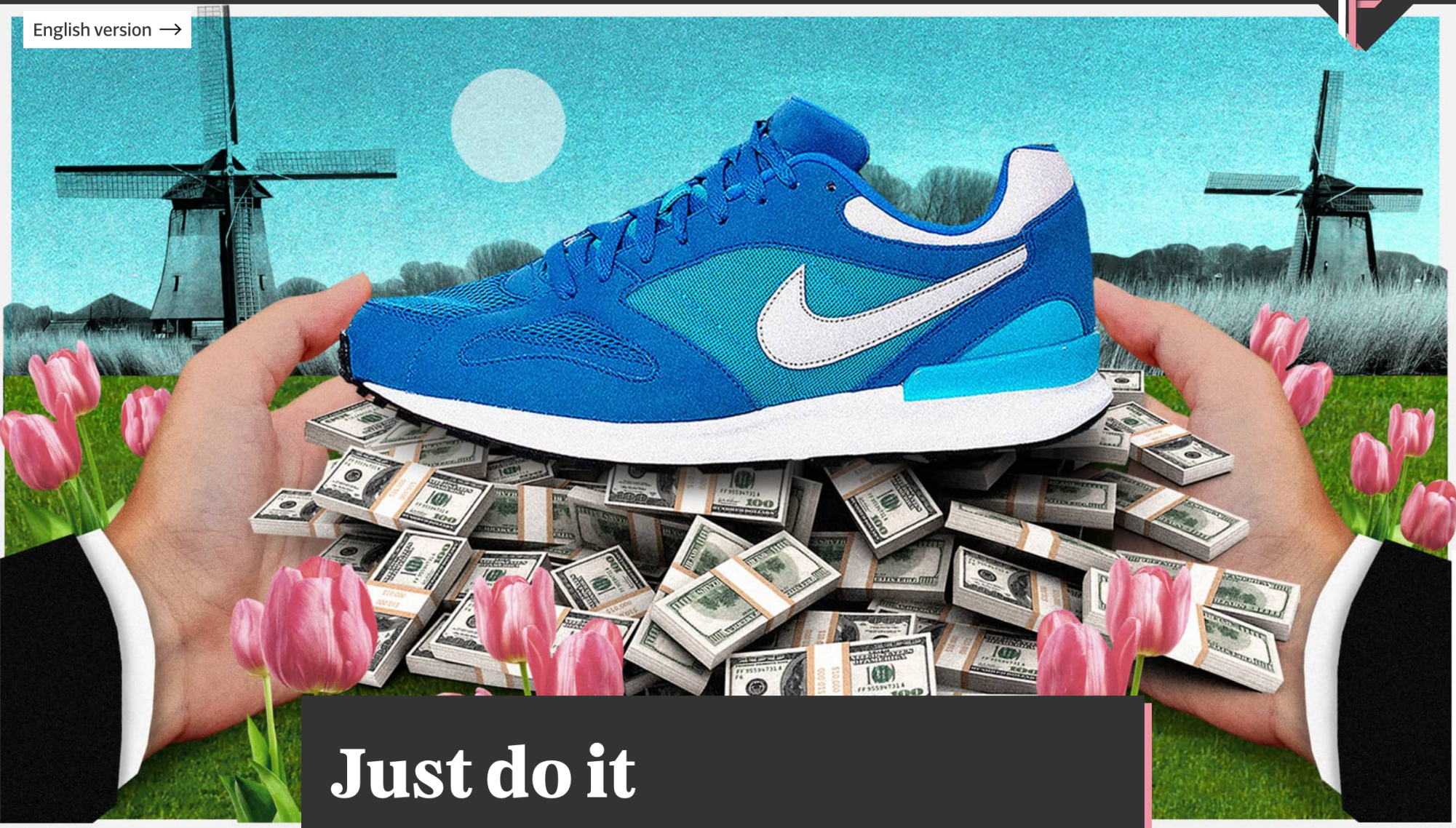
Im Mittel sind die Jahresdurchschnittstemperaturen hierzulande um 1,37 Grad Celsius angestiegen. Das zeigt die rote gerade Linie oben. Die Temperatur ist ein guter Indikator, weil sie in Messungen und Computermodellen gut und relativ genau zu handhaben ist.

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



https://www.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/

Datenjournalismus

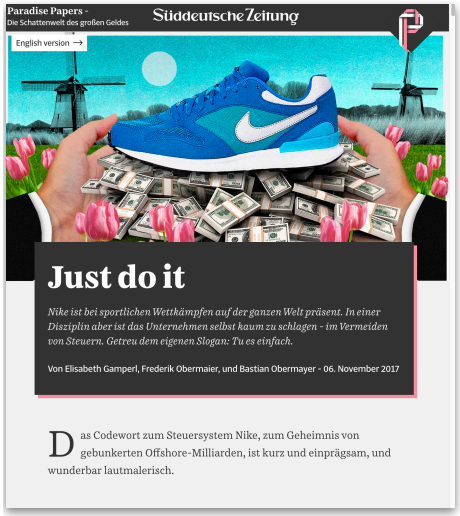
Datenjournalismus

Erstellung von Artikeln mit Hilfe von Datenanalysewerkzeugen bzw. das (teilweise) Einbinden dieser Werkzeuge in den Artikel selbst.

Einbinden von Datenanalyse in den Text?

Moment, das hatten wir schon mal:

Journalismus



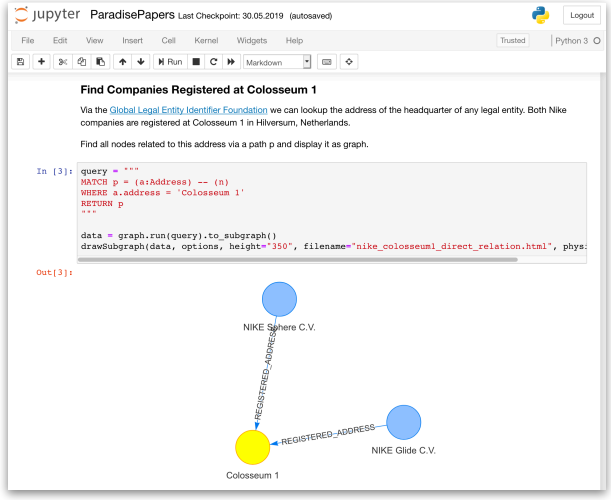
Software-Entwicklung

```
        return new_relation

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

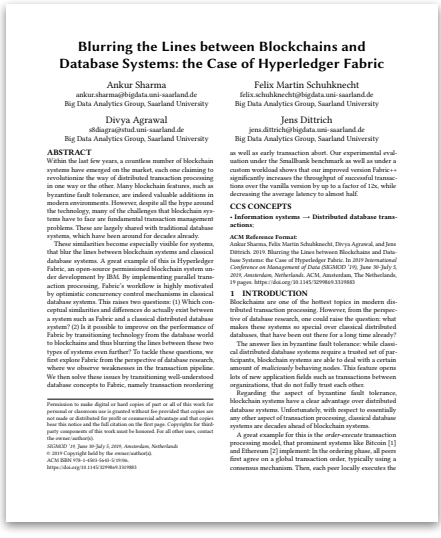
Hypertext+Code (z.B. Jupyter)

nähert sich an

nähert sich an



nähert sich an



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

Forschung

Datenjournalismus

2. Was sind die Datenmanagement und -analyseprobleme dahinter?

Frage 1

Wie verwalten wir graphische Daten?

Frage 2

Wie stellen wir Anfragen an diese Daten?

nächste Woche:

Frage 3

Wie visualisieren wir die Daten? Wie interagieren Nutzerinterface und DBMS?

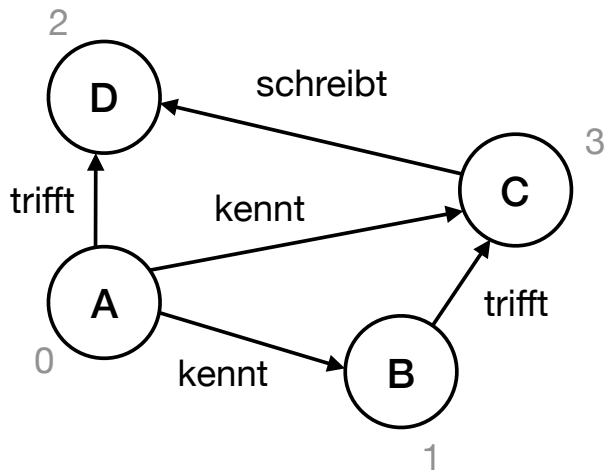
Datenjournalismus

3. Grundlagen, um diese Probleme lösen zu können
   (a) Folien
   (b) Jupyter/Python/SQL Hands-on

- Graphisches Datenmodell: nativ vs relationales Modell
- Anfragen an Graphen: SQL vs Cypher

Ein Sozialer Graph im Relationalen Model

Beispielgraph:



Adjazenzmatrix:

		nach			
		A_0	B_1	D_2	C_3
von	A_0		kennt	trifft	kennt
	B_1				trifft
	D_2				
	C_3			schreibt	

- Die Adjazenzmatrix des ungerichteten Graphen erinnert bereits an eine Relation.
- Allerdings ist dies ein Spezialfall einer Relation: eine Pivottabelle: Attribute stehen sowohl in den Spalten als auch in den Zeilen.

Pivottabellen

Adjazenzmatrix (als Relation):

von	nach	Label
A_0	B_1	kennt
A_0	D_2	trifft
A_0	C_3	kennt
B_1	C_3	trifft
C_3	D_2	schreibt

Adjazenzmatrix (Eine Pivottabelle):

von \ nach	A_0	B_1	D_2	C_3
A_0		kennt	trifft	kennt
B_1				trifft
D_2				
C_3			schreibt	

Pivottabellen

In einer Pivottabelle werden aus einer zugrundeliegenden Relation $x \geq 1$ Attribute in den Spalten und $y \geq 1$ Attribute in den Zeilen dargestellt. Am Kreuzungspunkt jedes Spalten-/Zeilenpaars werden $z \geq 1$ Attribute des zugrundeliegenden Tupels der Relation dargestellt.

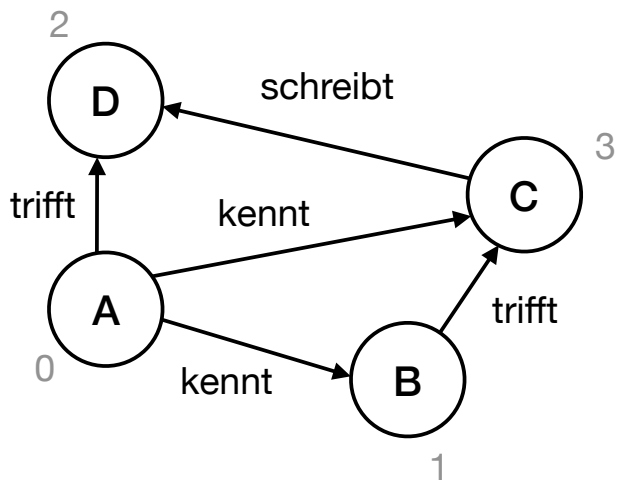
in SQL gibt es dazu den Befehl `PIVOT`, in PostgreSQL `\crosstabview`

Pivottabellen: Anmerkungen

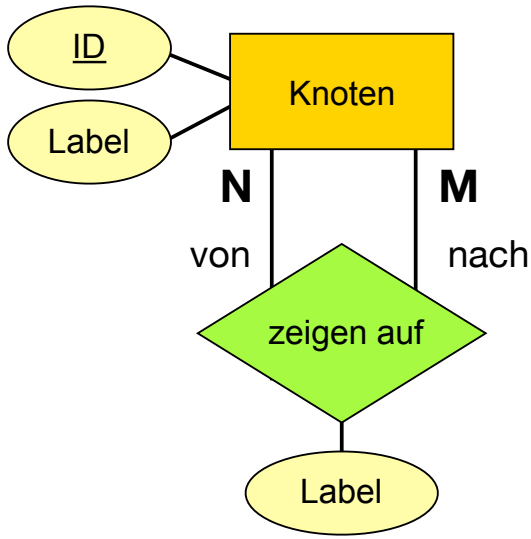
- falls die zugrundeliegende Relation nicht vorher bereits über alle Attribute in x **und** y gruppiert und aggregiert wurde, können in einer Zelle potentiell mehrere Tupel auftauchen
- deswegen wird meistens vorher aggregiert
- dies ist aber kein Muss
- es können auch mehrere Tupel pro Zelle angezeigt werden bzw. geeignet visualisiert werden

Ein Graph im Relationalen Modell: erster Ansatz

Beispielgraph:



ER-Modell:



Relationales Modell:

[Knoten] : {[ID:int, Label:str]}
[zeigen_auf] : {[
von_ID:(Knoten→ID),
nach_ID:(Knoten→ID),
Label:str
]}

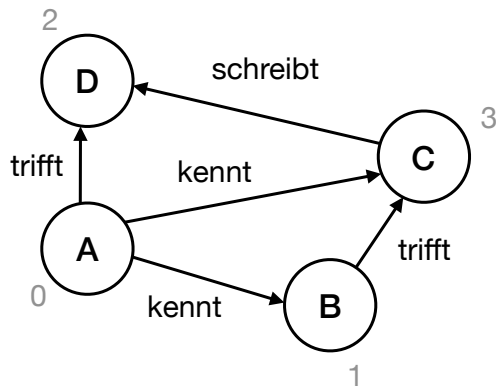
Beispieldaten:

Knoten	
ID	**Label**
0	A
1	B
2	D
3	C

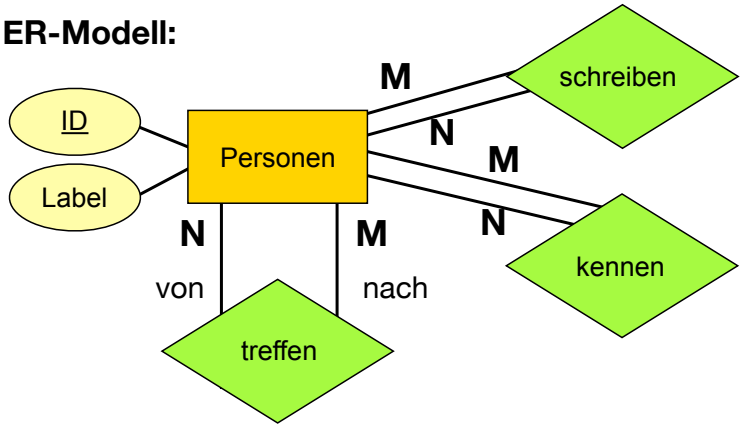
zeigen_auf		
von_ID	**nach_ID**	**Label**
0	2	trifft
0	1	kennt
0	3	kennt
1	3	trifft
3	2	schreibt

Ein Graph im Relationalen Modell: zweiter Ansatz

Beispielgraph:



ER-Modell:



Relationales Modell:

[Personen] : {[ID:int, Label:str]}
[treffen] : {[
    von_ID:(Personen→ID),
    nach_ID:(Personen→ID)
]}
[kennen] : {[
    von_ID:(Personen→ID),
    nach_ID:(Personen→ID)
]}
usw.

Beispieldaten:

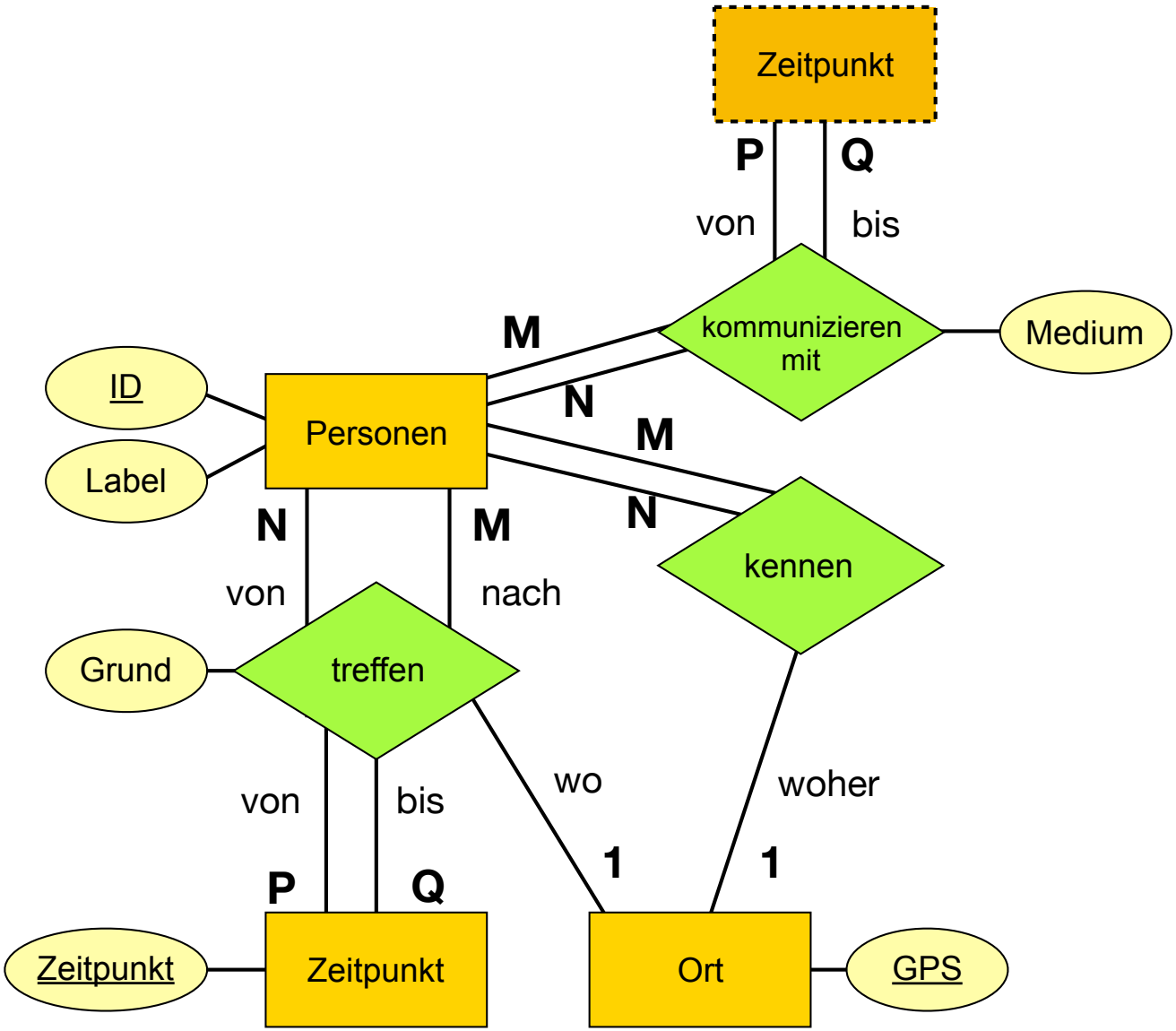
Personen	
ID	**Label**
0	A
1	B
2	D
3	C

treffen	
von_ID	**nach_ID**
0	2
1	3

kennen	
von_ID	**nach_ID**
0	1
0	3

schreiben	
von_ID	**nach_ID**
3	2

Ein Graph im Relationalen Modell: dritter Ansatz



Zwischenstand: Graphen im Relationalen Modell

Zwischenstand

Für die Modellierung/Speicherung von Graphen ist das relationale Modell perfekt geeignet. Wir können Graphen auf ganz unterschiedliche Arten in ER bzw. im relationalen Modell modellieren. Ein neues/separates Datenmodell ist **nicht** notwendig.

Zwischenstand:

- Graphisches Datenmodell: nativ vs relationales Modell

and the winner is:

das Relationale Modell

Wie sieht es mit Anfragen aus?

Da müssen wir etwas ausholen...

SQL: lokale Sichten mit `WITH` (Common Table Expressions)

Sichten (Views) in SQL mit `CREATE VIEW` hatten wir im SQL-Notebook.

lokale Sichten mit `WITH` (Common Table Expressions, CTEs)

Mit `WITH` wird eine lokale Sicht definiert, die nur innerhalb einer bestimmten SQL-Anfrage sichtbar ist. Diese lokale Sicht wird im Gegensatz zu `CREATE VIEW` nicht im System hinterlegt.

Globale Sicht:

nach Definition beliebig oft verwendbar

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

SELECT * FROM foo;
```

Ja, dies sind zwei Statements.

vs

Lokale Sicht:

nach Definition nur innerhalb dieser Anfrage verwendbar

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

Ja, dies ist nur ein Statement.

Mehr zur WITH-Syntax

Die Spalten einer lokalen Sicht können benannt werden:

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

ist äquivalent zu:

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

Im folgenden Beispiel sind die Attribute der Ausgaberelation nicht benannt worden:

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

Ausgabe in PostgreSQL:

?column?

	?column? integer	?column? integer
1	1	42

WITH RECURSIVE-Semantik

Prinzip:

```
WITH RECURSIVE foo AS (
      nichtRekursiverTerm()
  UNION
      rekursiverTerm(foo)
)
SELECT * FROM foo;
```

Algorithmus:

tmp = nichtRekursiverTerm()
foo = tmp
Solange tmp $\neq \{\}$:
    tmp = rekursiverTerm(tmp)
    foo $\cup=$ tmp
return foo

- UNION: Mengensemantik für den gesamten Algorithmus
- UNION ALL: **Multi**mengensemantik für den gesamten Algorithmus
- siehe PostgreSQL Doku zu WITH/Common Table Expressions

WITH RECURSIVE-Beispiel mit UNION

Beispielanfrage:

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

Algorithmus:

tmp = nichtRekursiverTerm()
foo = tmp
Solange tmp ≠ {}:
    tmp = rekursiverTerm(tmp)
    foo ∪= tmp
return foo

Algorithmus (Trace):

tmp = {(1)}
foo = tmp

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

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

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

% Solange: tmp ≠ {}: Abbruch der Schleife

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

WITH RECURSIVE-Beispiel mit UNION ALL

Beispielanfrage:

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

Algorithmus:

tmp = nichtRekursiverTerm()
foo = tmp
Solange tmp $\neq \{\}$:
    tmp = rekursiverTerm(tmp)
    foo $\cup$= tmp
return foo

Algorithmus (Trace):

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

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

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

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

% Solange: tmp $\neq \{\}$: Abbruch der Schleife

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

Graphen mit 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)
}
```

siehe: Graphs in SQL.ipynb

Zwischenstand: SQL als Anfragesprache für Graphen

Zwischenstand

Für sehr einfache Anfragen auf Graphen ist SQL durchaus geeignet; für komplexere Anfragen allerdings nur bedingt. Rekursive Anfragen in SQL werden schnell schwer lesbar (und damit schwer wartbar).

Was sind die Alternativen?

Cypher und Neo4j

Cypher

Cypher ist eins (von vielen) Beispielen für eine **domänspezifische Anfragesprache**. Cypher ist spezialisiert auf Graphen und wird aktuell (erst) von wenigen Systemen unterstützt.

Cypher:

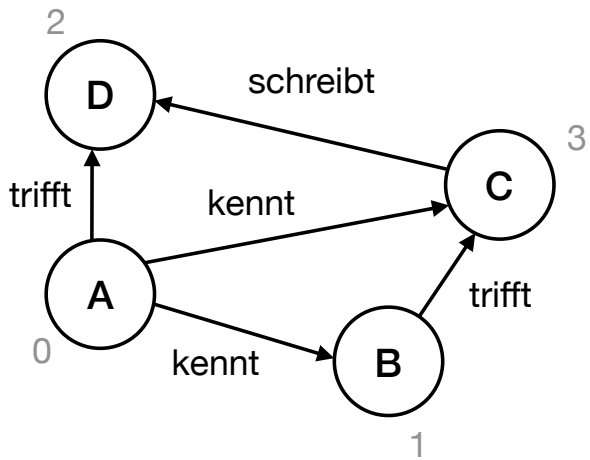
- seit 2011 entwickelt
- wird Teil von Apache 3.0
- Open Cypher (seit 2015): `https://www.opencypher.org`
- Cypher-Tutorial: `https://neo4j.com/developer/cypher-query-language/`
- Details zur Sprachdefinition: Cypher: An Evolving Query Language for Property Graphs, SIGMOD 2018

Neo4j:

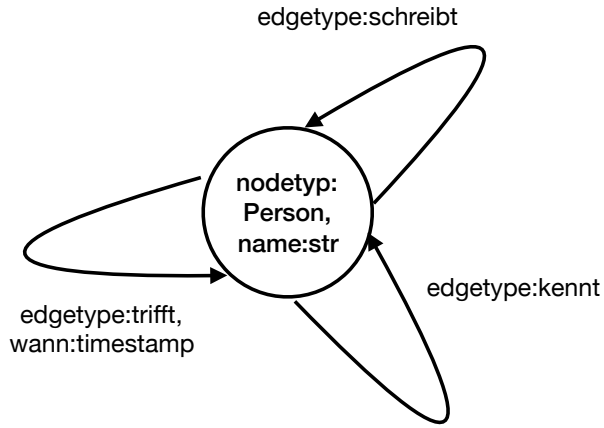
- open source und kommerzielle Lizenz
- `https://github.com/neo4j`
- `https://neo4j.com/`

Graphschema in Neo4j?

Beispielgraph (Instanz):



Beispielgraph („Graphschema“):



(fast) Schemafreie Graphen

Ein echtes Schema so wie in SQL lässt sich in Neo4j leider nicht definieren.

Warum „fast“? Was wir einschränken können ist:

- Knoten und Kanten sind typisiert
- ein bestimmtes Attribut muss existieren (andere Attribute können aber beliebig hinzugefügt werden)
- Schlüssel über Attribute (ähnlich wie in SQL)
- siehe Neo4j-Doku: 5.1 Constraints

Grundstruktur von Cypher-Anfragen: MATCH und RETURN

Im Folgenden gibt es Knoten vom Typ PERSON und Kanten der Typen WRITES, MEETS, KNOWS.

MATCH und RETURN

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

Gibt alle Knoten vom Typ PERSON zurück und zeigt deren Attribute node_id und name an.

Ergebnis:

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

- MATCH (n:PERSON) entspricht FROM person AS n in SQL
- RETURN entspricht SELECT in SQL
- () ist die sogenannte ASCII-Anfragesyntax für einen Knoten

ASCII-Anfragesyntax: ()-[]->()

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

Gibt alle Kanten vom Typ KNOWS zurück. Die Richtung der Kante wird hierbei berücksichtigt.

Ergebnis:

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

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

Gibt alle Kanten vom Typ KNOWS zurück. Die Richtung der Kante wird hierbei **nicht** berücksichtigt. Somit wird jede matchende Kante zweimal ausgegeben.

Ergebnis:

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

WITH

WITH in Cypher ist sehr ähnlich zu WITH in SQL und definiert auch in Cypher eine lokale Sicht.

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

Gibt alle Zyklen bis maximaler Länge 6 zurück.

- NODES() extrahiert aus jedem mit MATCH selektierten Pfad die enthaltenen Knoten als Liste
- diese Liste kann per index referenziert werden: path[0]

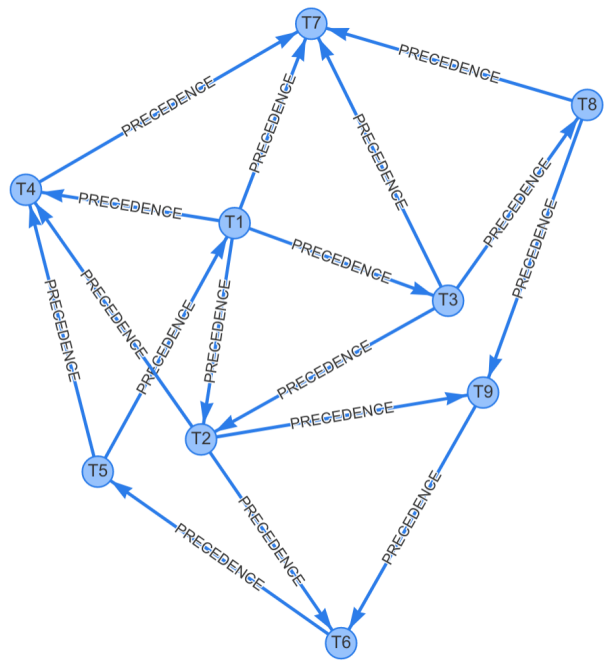
Graphen mit Cypher (Graphs in Cypher.ipynb)

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

Zwischenstand

Zwischenstand: Anfragen an Graphen in SQL vs Cypher

Für Anfragen an Graphen ist Cypher eine schlanke und elegante Sprache und deutlich besser geeignet als SQL.

Zwischenstand:

- Cypher vs SQL

and the winner is:

Cypher

Gesamtergebnis

- Graphisches Datenmodell: nativ vs relationales Modell

and the winner is:

das Relationale Modell

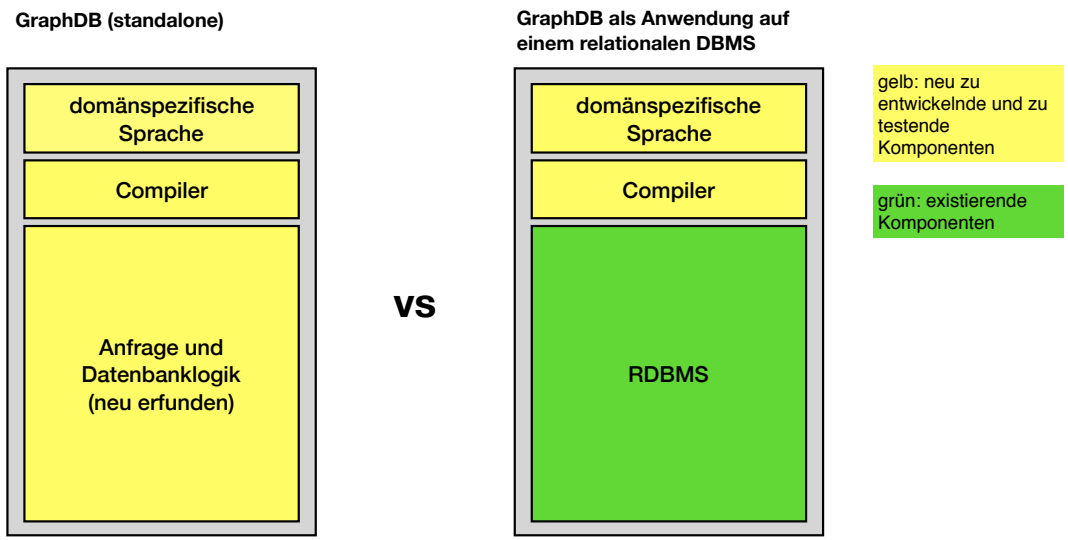
- Cypher vs SQL

and the winner is:

Cypher

Die interne Architektur einer „Graphdatenbank“

- Wie Neo4j intern die Daten ablegt, ist unklar.
- Die technisch beste Entscheidung hier wäre/ist aber, Cypher als Anwendung auf einem relationalen DBMS zu konzipieren.
- D.h. anstatt ein „Graphdatenbanksystem“ von Grund auf neu zu entwickeln, wäre/ist es besser, sich auf funktionierende, etablierte und getestete Techniken zu stützen.



Diesen Design-Fehler sieht man leider sehr häufig bei der Entwicklung „neuer“ Datenbanksysteme.

2020

agensgraph
sqlserver

Datenjournalismus

nächstes Mal:

4. Transfer der Grundlagen auf die konkrete Anwendung