

# Hausarbeit 30100 Big Data

---



Hochschule  
Albstadt-Sigmaringen  
Albstadt-Sigmaringen University

**Autor:** Kevin Südmersen

**Datum:** 24.05.2021

# Inhaltsangabe

---

## **Hausarbeit 30100 Big Data**

### **Inhaltsangabe**

#### **Hadoop, Hive, Spark**

##### Übung 2.1

- (1) Über welches Protokoll werden die Dateiblöcke verteilt?
- (2) Wie viele Mapper gibt es auf welchen Nodes?
- (3) Wie viele Dateiblöcke enthält jeder Node?

##### Übung 2.2

##### Übung 2.3

##### Übung 2.4

##### Übung 2.5

##### Übung 2.6

##### Übung 2.7

Tägliche unit\_sales

Wöchentliche unit\_sales

##### Übung 2.9

#### **Verteilte relationale DBMS**

##### **MongoDB**

##### Teilaufgabe 1

##### Teilaufgabe 2

Lösungsweg 1

Lösungsweg 2

##### Teilaufgabe 3

##### **Neo4j**

# Hadoop, Hive, Spark

---

## Übung 2.1

---

Ein Hadoopcluster besteht aus 4 DataNodes mit den Parametern `blocksize` 256 MB und `splitsize` 512 MB. Es soll die Datei `kfz.txt` der Größe 1 TB verteilt werden.

### (1) Über welches Protokoll werden die Dateiblöcke verteilt?

SSH (Secure Shell)

### (2) Wie viele Mapper gibt es auf welchen Nodes?

Ein Mapper bearbeitet einen Split. Ein Split besteht aus  $512 / 256 = 2$  Blöcken, also bearbeitet ein Mapper 2 Blöcke

Es gibt  $1 \text{ TB} / 512 \text{ MB} = 2$  Millionen Splits, die auf 4 Nodes verteilt sind, also auf jeder Node gibt es 500.000 Splits und deshalb 500.000 Mapper pro Node.

### (3) Wie viele Dateiblöcke enthält jeder Node?

Es gibt  $1 \text{ TB} / 256 \text{ MB} = 4$  Millionen Blöcke, die auf 4 Nodes verteilt sind, also enthält jede Node 1 Millionen Blöcke

## Übung 2.2

---

Welche Ausgabedaten liefern die Prozesse Map, Shuffle und Sort & Reduce für das SELECT-Statement `SELECT count(identnr), identnr FROM kfz GROUP BY identnr` ?

Map Prozess

- Eingabe: Datei `kfz.txt`
- Ausgabe: Liste von Tupeln mit folgenden Key, Value `(identnr, 1)` Paaren: `[(1, 1), (1, 1), (2, 1), (1, 1)]`

Shuffle & Sort Prozess

- Eingabe: Key, Value Paare vom Map Prozess
- Sortiert und gruppiert nach `identnr`, also erzeugt dabei folgende Gruppen
  - `group(identnr == 1) = [(1, 1), (1, 1), (1, 1)]`
  - `group(identnr == 2) = [(2, 1)]`
- Ausgabe: 1 Datei pro `identnr`

Reduce Prozess

- Eingabe: Jeder Reduce Prozess bekommt eine Datei/Gruppe von der Ausgabe des Shuffle & Sort Prozesses
- Jeder Reducer berechnet die Summe der Values jeder Gruppe
- Ausgabe: 1 Datei mit den Spalten `count(identnr)` und `identnr`

## Übung 2.3

---

Um die SQL Abfragen dieser Aufgabe ausführen zu können, muss eine Tabelle mit Namen `verkaufteartikel` in Hive existieren. Um die Daten dieser Hive Tabelle in mein lokal installiertes Hadoop Cluster zu transferieren, habe ich im HDFS des Kubernetes Cluster der Hochschule nach einer Datei `verkaufteartikel` mittels `hadoop fs -find / -name "verkaufteartikel"` gesucht. Danach habe ich die gefundenen Dateipfade mittels `hadoop fs -copyToLocal <location_of_verkaufteartikel_in_hdfs> <desired_location_on_host>` auf den Host des Hadoop Clusters kopiert, und danach habe ich die dazugehörigen Daten mittels WinSCP auf meinen lokalen Rechner kopiert.

Die Daten in `verkaufteartikel` sehen folgendermaßen aus:

```
2,2016-12-03,3
1,2017-04-17,24
2,2018-05-07,17
3,2019-09-12,33
4,2020-12-20,14
```

Diese Daten habe ich nun über ein Volume in die Namenode meines lokal installierten Hadoop Clusters kopiert und habe auf der Kommandozeile der Namenode den Befehl `hadoop fs -mkdir -p hadoop-data/verkaufteartikel` ausgeführt, um das Verzeichnis `hadoop-data/verkaufteartikel` im HDFS zu erzeugen. Danach habe ich mittels `hadoop fs -copyFromLocal verkaufteartikel.csv hadoop-data/verkaufteartikel` die Daten in das gerade erzeugte Verzeichnis kopiert.

Danach habe ich einen SQL Query Editor in dem Hue Dienst (Hue ist ein Cluster Management Dienst so ähnlich wie Ambari) geöffnet und mit dem SQL Statement

```
-- Convert verkaufteartikel into a Hive table
CREATE EXTERNAL TABLE IF NOT EXISTS verkaufteartikel (
  id INT,
  date_ DATE,
  quantity INT
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
LOCATION '/user/root/hadoop-data/verkaufteartikel';
```

die Daten von `verkaufteartikel` in eine externe Hive Tabelle geladen.

Nun konnte ich endlich die SQL Statements aus der Übungsaufgabe ausführen. Das Ergebnis der ersten SQL Abfrage

```
EXPLAIN SELECT *
FROM verkaufteartikel
WHERE date_ > '2017-01-01';
```

ist wie folgt:

```
Explain
STAGE DEPENDENCIES:
  Stage-0 is a root stage

STAGE PLANS:
```

```

Stage: Stage-0
  Fetch Operator
    limit: -1
    Processor Tree:
      TableScan
        alias: verkaufteartikel
        Statistics: Num rows: 1 Data size: 79 Basic stats: COMPLETE Column
stats: NONE
      Filter Operator
        predicate: (date_ > 2017-01-01) (type: boolean)
        Statistics: Num rows: 1 Data size: 79 Basic stats: COMPLETE Column
stats: NONE
      Select Operator
        expressions: id (type: int), date_ (type: date), quantity (type:
int)
        outputColumnNames: _col0, _col1, _col2
        Statistics: Num rows: 1 Data size: 79 Basic stats: COMPLETE Column
stats: NONE
      ListSink

```

- `TableScan` bedeutet, dass jede Zeile von `verkaufteartikel` einmal in den Hauptspeicher eingelesen werden musste. Natürlich sollten nicht alle Zeilen auch im Hauptspeicher verbleiben, wenn man mit großen Datenmengen zu tun hat.
- Der `Filter operator` kommt durch die `WHERE` Klausel im SQL Statement zustande und behält nur die Zeilen von `verkaufteartikel`, die die dazugehörige Bedingung erfüllen.
- `predicate (date_ > 2017-01-01)` ist das zu der `WHERE` Klausel gehörende Prädikat, was immer `true` zurückgibt, wenn die Bedingung erfüllt ist, und `false` ansonsten.
- `Select operator` ist eine Projektion auf gewisse Spaltennamen, in unserem Fall wurden mittels `*` alle Spaltennamen selektiert, und deshalb sind in `expressions` alle Spaltennamen aufgeführt.

Nun zur 2. SQL Abfrage. Das Ergebnis der Abfrage

```

EXPLAIN SELECT date_, count(*)
FROM verkaufteartikel
GROUP BY date_;

```

ist folgendes:

```

Explain
STAGE DEPENDENCIES:
  Stage-1 is a root stage
  Stage-0 depends on stages: Stage-1

STAGE PLANS:
  Stage: Stage-1
    Map Reduce
      Map Operator Tree:
        TableScan
          alias: verkaufteartikel
          Statistics: Num rows: 1 Data size: 79 Basic stats: COMPLETE Column
stats: NONE
        Select Operator
          expressions: date_ (type: date)
          outputColumnNames: date_

```

```

Statistics: Num rows: 1 Data size: 79 Basic stats: COMPLETE Column
stats: NONE
  Group By Operator
    aggregations: count()
    keys: date_ (type: date)
    mode: hash
    outputColumnNames: _col0, _col1
    Statistics: Num rows: 1 Data size: 79 Basic stats: COMPLETE
Column stats: NONE
  Reduce Output Operator
    key expressions: _col0 (type: date)
    sort order: +
    Map-reduce partition columns: _col0 (type: date)
    Statistics: Num rows: 1 Data size: 79 Basic stats: COMPLETE
Column stats: NONE
    value expressions: _col1 (type: bigint)
  Reduce Operator Tree:
    Group By Operator
      aggregations: count(VALUE._col0)
      keys: KEY._col0 (type: date)
      mode: mergepartial
      outputColumnNames: _col0, _col1
      Statistics: Num rows: 1 Data size: 79 Basic stats: COMPLETE Column
stats: NONE
    File Output Operator
      compressed: false
      Statistics: Num rows: 1 Data size: 79 Basic stats: COMPLETE Column
stats: NONE
      table:
        input format: org.apache.hadoop.mapred.SequenceFileInputFormat
        output format:
org.apache.hadoop.hive ql.io.HiveSequenceFileOutputFormat
        serde: org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe

Stage: Stage-0
  Fetch Operator
    limit: -1
  Processor Tree:
    ListSink

```

- **Map Reduce Tree**
  - Hier wird wieder zuerst ein **TableScan** gemacht, und es wird auf die Spalte **date\_** projiziert, da dies die einzige Spalte ist, die man braucht, um das Ergebnis der Abfrage zu bekommen
  - Im **Group By Operator** (Shuffle & Sort Prozess) wird u.a. nach der **date\_** Spalte gruppiert. Pro Datum gibt es eine Gruppe. In unserem Fall gibt es genau 5 Gruppen, die jeweils ein einziges Datum beinhalten, nämlich:
    - 2016-12-03
    - 2017-04-17
    - 2018-05-07
    - 2019-09-12
    - 2020-12-20
- **Reduce Operator Tree**

- Hier wird für jede der obigen Gruppen nun die Aggregatsfunktion `count(VALUE._co10)` angewendet

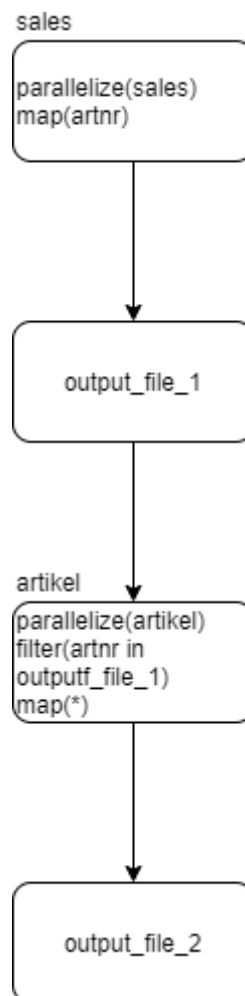
## Übung 2.4

Hadoop verteilt Dateien und Spark verteilt Programme, also auch SQL Abfragen, insbesondere JOINS. Aus dem Programm wird ein Directed Acyclic Graph (DAG) generiert und es wird versucht diesen DAG zu parallelisieren. Ein DAG ist ein Berechnungsgraph, der ein Anfang und ein Ende hat (also keine Zyklen), der den Programmablauf darstellt und diesen ausführt.

Der DAG zu der SQL Abfrage

```
SELECT * FROM artikel WHERE artnr IN (SELECT artnr FROM sales);
```

sieht folgendermaßen aus:



Zuerst wird die Subquery `SELECT artnr FROM sales` ausgeführt, die Ergebnismenge in der Datei `output_file_1` zwischengespeichert, und dann werden nur die Artikel aus der Tabelle `artikel` genommen, die in `output_file_1` vorkommen.

## Übung 2.5

Der Code mit Erklärungen befindet sich in meinem privaten [GitHub Repository](#) unter [jupyter-spark/work/assignments/uebung\\_25\\_rjdbc\\_hive.ipynb](#).

## Übung 2.6

Der Code mit Erklärungen befindet sich in meinem privaten [GitHub Repository](#) unter [jupyter-spark/work/assignments/uebung\\_26\\_pyspark.ipynb](#).

## Übung 2.7

Zu den Dateien von Übung 2.5 und Übung 2.6 sollen einige Auswertungen über Hive erzeugt werden. Da hier keine genauen Vorgaben gegeben sind, werde ich zuerst die täglichen `unit_sales` und danach die wöchentlichen Transaktionsvolumina bestimmen.

### Tägliche `unit_sales`

Wie bereits in anderen Übungen beschrieben habe ich zuerst die Dateien `holiday_events.csv`, `items.csv`, `quito_stores_sample2016_2017.csv` und `transactions.csv` in den namenode Container, dann in das HDFS und dann mittels dem Hue UI in Hive hinein geladen. Folgendes HiveQL Statement soll die täglichen `unit_sales` berechnen:

```
select sum(unit_sales) as sum_unit_sales, year(date_quito) as current_year,
month(date_quito) as current_month, day(date_quito) as current_day
from quito_stores_sample2016_2017
where date_format(date_quito, 'u') = 4
group by year(date_quito), month(date_quito), day(date_quito)
order by current_year, current_month, current_day;
```

Output:

sum_unit_sales	current_year	current_month	current_day
138728.353000000012	2016	8	18
136600.037999999977	2016	8	25
162824.917999999968	2016	9	1
151830.446999999987	2016	9	8
138104.188000000001	2016	9	15
126704.193999999986	2016	9	22
130462.7680000000001	2016	9	29
137414.1	2016	10	6
136179.534000000022	2016	10	13
150390.177999999972	2016	10	20
130887.8400000000003	2016	10	27

In Retrospektive, kam mir obiges Statement ein bisschen umständlich vor (weil zuerst das Jahr, der Monat und der Tag extrahiert, und danach wieder nach Jahr, Monat und Tag gruppiert werden muss), habe ich im folgenden Statement wieder die Summe der `unit_sales` berechnet, aber dieses mal habe ich nach dem Datum gruppiert.

```
select sum(unit_sales) as sum_unit_sales, date_quito
from quito_stores_sample2016_2017
where date_format(date_quito, 'u') = 4
group by date_quito
order by date_quito;
```

Output:



sum_unit_sales	date_quito
138728.35300000012	2016-08-18
136600.03799999977	2016-08-25
162824.91799999968	2016-09-01
151830.44699999987	2016-09-08
138104.18800000001	2016-09-15
126704.19399999986	2016-09-22
130462.768000000001	2016-09-29
137414.1	2016-10-06
136179.53400000022	2016-10-13
150390.17799999972	2016-10-20
130887.84000000003	2016-10-27

Der Output beider Statements sind identisch.

## Wöchentliche unit\_sales

Zuerst habe ich mich von der Vorlesung inspirieren lassen und habe die wöchentlichen Transaktionsvolumina folgendermaßen berechnet:

```
select weekofyear(tr.date_trans) as week, sum(tr.transactions) as
weekly_unit_sales
from items inner join quito_stores_sample2016_2017 AS quito_store on
quito_store.item_nbr_quito = items.item_nbr_item
inner join transactions AS tr on tr.store_nbr_trans =
quito_store.store_nbr_quito
inner join holidays_events on holidays_events.datum_holi = tr.date_trans
group by weekofyear(tr.date_trans)
order by week;
```

Ergebnis:

week	weekly_unit_sales
1	28567434275
2	9830893906
6	8730851354
7	8149803877
8	8729617503
9	27677998583
10	13910862599
12	4080833575
13	10579847436
14	17302118505
15	54653669532
16	57646139981
17	42643974148
18	68828867443
19	88542663017
20	11272350430
21	32300823752
24	8108878613
25	46405889614
26	62945616169
27	51432764895
28	17027589597
29	19450255582

```
30 46223357002
31 9531701329
32 38449772542
33 26209142984
34 13882155606
35 3979359634
39 14157756766
40 13135696715
41 32857151812
44 35150332788
45 66891536748
46 28736604080
47 5014115514
48 13026132482
49 57633576346
50 8591347588
51 77485626478
52 126260907220
53 5832310146
```

Nach einiger Überlegung dachte ich mir jedoch, dass man lediglich die Spalten `date_trans` und `transactions` aus der Tabelle `transactions` braucht, um die wöchentlichen Transaktionsvolumina zu berechnen. Deshalb hatte ich die JOINS aus dem obigen Statement herausgenommen und das Statement noch einmal ausgeführt:

```
select weekofyear(date_trans) as week, sum(transactions) as weekly_unit_sales
from transactions
group by weekofyear(date_trans)
order by week;
```

Ergebnis:

week	weekly_unit_sales
1	2904264
2	2876856
3	2828681
4	2787688
5	2890639
6	2914532
7	2944731
8	2836890
9	2947415
10	2967329
11	2889503
12	2872067
13	2893670
14	2999954
15	2919038
16	2997701
17	2884039
18	3057556
19	3031439
20	2891410
21	2835168
22	3013612
23	2966299

```
24 2925681
25 2861633
26 2848133
27 2998279
28 2852663
29 2851306
30 2847369
31 2986859
32 2881863
33 2472897
34 2272435
35 2312863
36 2413073
37 2282162
38 2258478
39 2231450
40 2364076
41 2279355
42 2246341
43 2205255
44 2256071
45 2348502
46 2265909
47 2284200
48 2321219
49 2479496
50 2558796
51 3016656
52 2885163
53 520281
```

Wie man jetzt jedoch sehen kann sind alle Transaktionsvolumina sehr viel kleiner. Dies liegt daran, dass durch die vielen JOINS im ersten HiveQL Statement die Transaktionsvolumina künstlich aufgebläht werden, da es scheinbar in den an den JOINS beteiligten Tabellen duplikate JOIN Partner gibt. Z.B. werden die Tabellen `transactions` und `quito_store` folgendermaßen ge-join: `inner join transactions AS tr on tr.store_nbr_trans = quito_store.store_nbr_quito`. Wenn es zwischen diesen beiden Tabellen duplikate JOIN Partner gibt, dann muss es für manche `tr.store_nbr_trans` mehr als einen korrespondierenden `quito_store.store_nbr_quito` geben. Aufgrund dieser künstlich aufgeblähten Transaktionsvolumina, würde ich persönlich das zweite HiveQL Statement bevorzugen.

## Übung 2.9

Der Code mit Erklärungen befindet sich in meinem privaten [GitHub Repository](#) unter [jupyter-spark/work/assignments/uebung\\_29\\_pyspark.ipynb](#).

# Verteilte relationale DBMS

Ein bestimmte SQL Abfrage soll entweder mithilfe von AWS Redshift oder MS SQL Server ausgeführt werden. Ich habe mich dazu entschlossen folgendes SQL Statement in Amazon Redshift auszuführen,

```
select
    referenzdatum,
    bundesland,
    landkreis,
    -- Get the average of the last 7 days (relative to the current
    `referenzdatum`)
    -- in the current bundesland and landkreis
    (select avg(infiziert) as durchschnitt
     from vcoronaerkrankung vc2
     where vc2.referenzdatum <= vc1.referenzdatum
     and vc2.referenzdatum > (vc1.referenzdatum - 7)
     and vc2.bundesland = vc1.bundesland
     and vc2.landkreis = vc1.landkreis)
from vcoronaerkrankung vc1;
```

welches folgende Ergebnismenge (insgesamt 22761 Zeilen) zurückliefert:

referenzdatum	bundesland	landkreis	durchschnitt
2021-02-15	Sachsen	LK Bautzen	35
2021-02-15	Sachsen	LK Mittelsachsen	29
2021-02-15	Bayern	SK Augsburg	21
2021-02-15	Mecklenburg-Vorpommern	SK Rostock	7
2021-02-15	Sachsen	LK Vogtlandkreis	48
2021-02-15	Thüringen	LK Unstrut-Hainich-Kreis	25
2021-02-15	Nordrhein-Westfalen	LK Herford	14
2021-02-15	Hessen	LK Kassel	13
2021-02-15	Bayern	SK Regensburg	5
2021-02-15	Bayern	SK Schweinfurt	1
2021-02-15	Niedersachsen	LK Osnabrück	51
2021-02-15	Nordrhein-Westfalen	LK Borken	31
2021-02-15	Nordrhein-Westfalen	SK Mönchengladbach	9
2021-02-15	Rheinland-Pfalz	LK Bad Kreuznach	11
2021-02-15	Bremen	SK Bremerhaven	17
2021-02-15	Baden-Württemberg	LK Rems-Murr-Kreis	24
2021-02-15	Bayern	LK Dachau	9
2021-02-15	Bayern	LK Erlangen-Höchstadt	5
2021-02-15	Thüringen	LK Saale-Orla-Kreis	20
2021-02-15	Rheinland-Pfalz	SK Trier	5

Obige Ergebnismenge soll die durchschnittliche Anzahl an Infektionen innerhalb der letzten 7 Tage (relativ zu einem bestimmtem Referenzdatum) für ein gewissen Landkreis in einem gewissen Bundesland zeigen.

# MongoDB

Zuerst habe ich versucht die Datei `listingsAndReviews.json` mittels `docker exec mongo mongoimport --username=kevinsuedmersen --password=secret --host=mongo:27017 --db=airbnb --collection=listings_and_reviews --authenticationDatabase=admin --file=/mongo-data/airbnb/listingsAndReviews.json` in eine MongoDB Instanz in meinem lokalen `docker-compose` Netzwerk zu importieren, jedoch kamen dabei verschiedenste Importfehler, die wahrscheinlich damit zu tun hatten, dass manche Felder in `listingsAndReviews.json` Werte wie z.B. `NumberDecimal("1.0")` hatten, also Werte, die nicht durchgehend als Strings formatiert sind, wie es in `json` Dateien normalerweise üblich ist.

Deshalb habe ich mich mit meinem lokal installierten MongoDB Compass auf das MongoDB Cluster der Hochschule verbunden. Dabei musste ich lediglich den Connection String `mongodb+srv://thomas:Morgen0007@cluster1.u6ruv.mongodb.net/test` in Mongo Compass einfügen. Für alle folgenden Aufgaben habe ich als Basis die Daten in `sample_airbnb.listingsAndReviews` (also die Datenbank `sample_airbnb` und die Collection `listingsAndReviews`) verwendet.

## Teilaufgabe 1

Ermitteln Sie die Adresse mit dem höchsten Preis.

In dem `Aggregations` Tab von MongoDB Compass habe ich folgende Aggregation erzeugt:

The screenshot shows the MongoDB Compass interface with the `sample_airbnb.listingsAndReviews` collection selected. The `Aggregations` tab is active, displaying a pipeline with three stages:

- \$project**: Selects relevant fields.

```
{ "$project": { "price": 1, "address": 1 } }
```
- \$sort**: Sorts documents by price in descending order.

```
{ "$sort": { "price": -1 } }
```
- \$limit**: Limits the results to 1 document.

```
{ "$limit": 1 }
```

The output of the aggregation shows a single document with the highest price:

```
{ "_id": "10000000", "price": 10000.00, "address": "10000" }
```

die, wenn man sie in Python Code exportieren möchte folgendermaßen aussehen würde (exklusive der Kommentare):

```
[
    # select relevant fields
    {
        '$project': {
            'price': 1,
            'address': 1
        }
    },
    # sort by price in descending order
```

```

{
  '$sort': {
    'price': -1
  },
  # Select the first result, i.e. the document with the highest price
  {
    '$limit': 1
  }
}
]

```

## Teilaufgabe 2

Ermitteln Sie pro Adresse die Anzahl an `amenities`.

## Lösungsweg 1

Aggregation Pipeline in MongoDB Compass:

The screenshot shows the MongoDB Compass interface with an aggregation pipeline for the `sample_airbnb_listings_and_reviews` collection. The pipeline consists of four stages:

- \$project**: Projects the `listing_url`, `amenities` array, and `address` object.
- \$set**: Sets the field `n_amenities_per_listing` to the size of the `amenities` array using `$size: '$amenities'`.
- \$group**: Groups documents by `address` and calculates the sum of `n_amenities_per_listing` for each group.
- \$sort**: Sorts the results by `n_amenities_per_listing` in descending order.

The output of the pipeline shows the aggregated results for each address, with the number of amenities per listing summed up.

Pipeline exportiert nach Python Code:

```

[
  {
    '$project': {
      'address': 1,
      'amenities': 1
    },
    # Calculate the number of amenities per listing, i.e. for each document
    {
      '$set': {
        'n_amenities_per_listing': {
          '$size': '$amenities'
        }
      },
      # Group by address and calculate the sum of amenities per listing
      '$group': {
        '_id': '$address',

```

```

    'n_amenities_per_address': {
      '$sum': '$n_amenities_per_listing'
    }
  }
}
]

```

Hier wird in der `$set` Stage zuerst die Länge der `amenities` array pro Listing, also pro Dokument in der Collection `listingsAndReviews` bestimmt und als zusätzliches Feld `n_amenities_per_listing` hinzugefügt. Danach wird nach `address` gruppiert und `n_amenities_per_listing` aufsummiert, da es ja theoretisch sein könnte, dass manch eine `address` mehr als einmal in `listingsAndReviews` vorkommt.

## Lösungsweg 2

MongoDB Compass:

The screenshot shows the MongoDB Compass interface with the following aggregation pipeline:

```

[
  {
    '$set': {
      'n_amenities_per_listing': {
        '$size': '$amenities'
      }
    }
  },
  {
    '$group': {
      '_id': '$address',
      'n_amenities_per_address': {
        '$sum': '$n_amenities_per_listing'
      }
    }
  }
]

```

The output shows 20 documents, each representing an address and its total number of amenities.

Exportierter Python Code:

```

[
  {
    '$project': {
      'address': 1,
      'amenities': 1
    }
  },
  # Create one document per address AND amenity by unrolling the amenities array
  {
    '$unwind': {
      'path': '$amenities'
    }
  },
  # Group by address and count how many elements we have in each group
  {
    '$group': {
      '_id': '$address',
      'n_amenities': {
        '$sum': 1
      }
    }
  }
]

```

```

    }
  }
}
]

```

Hier wird die `amenities` array aufgerollt, d.h. dass als Zwischenergebnis der `unwind` Stage ein Dokument pro Adresse *und* Item in der `amenities` Array zurückkommt (siehe Screenshot). Wenn man nun anschließend nach `address` gruppiert und die Elemente in jeder Gruppe aufzählt, dann weiß man also wie viele `amenities` zu welcher `address` gehören.

## Teilaufgabe 3

Ermitteln Sie die Adresse mit den meisten `amenities`.

MongoDB Compass:

The screenshot shows the MongoDB Compass interface with an aggregation pipeline applied to the `sample_airbnb_listings_and_reviews` collection. The pipeline consists of four stages:

- \$set**: Adds a field `n_amenities_per_listing` to each document, representing the size of the `amenities` array.
- \$group**: Groups documents by `address` and calculates the sum of `n_amenities_per_listing` for each group, creating a new field `n_amenities_per_address`.
- \$set**: Adds a field `rank` to each document, representing the rank of the address based on the number of amenities.
- \$limit**: Limits the results to the top 10 addresses with the most amenities.

The output of the aggregation is displayed in the right-hand pane, showing the top 10 addresses and their corresponding `n_amenities_per_address` values.

Exportierter Python Code:

```

[
  # Based on 1. approach from Teilaufgabe 2
  {
    '$project': {
      'address': 1,
      'amenities': 1
    }
  }, {
    '$set': {
      'n_amenities_per_listing': {
        '$size': '$amenities'
      }
    }
  }, {
    '$group': {
      '_id': '$address',
      'n_amenities_per_address': {
        '$sum': '$n_amenities_per_listing'
      }
    }
  }, {
    '$set': {
      'rank': {
        '$rank': '$n_amenities_per_address'
      }
    }
  }, {
    '$limit': 10
  }
]

```



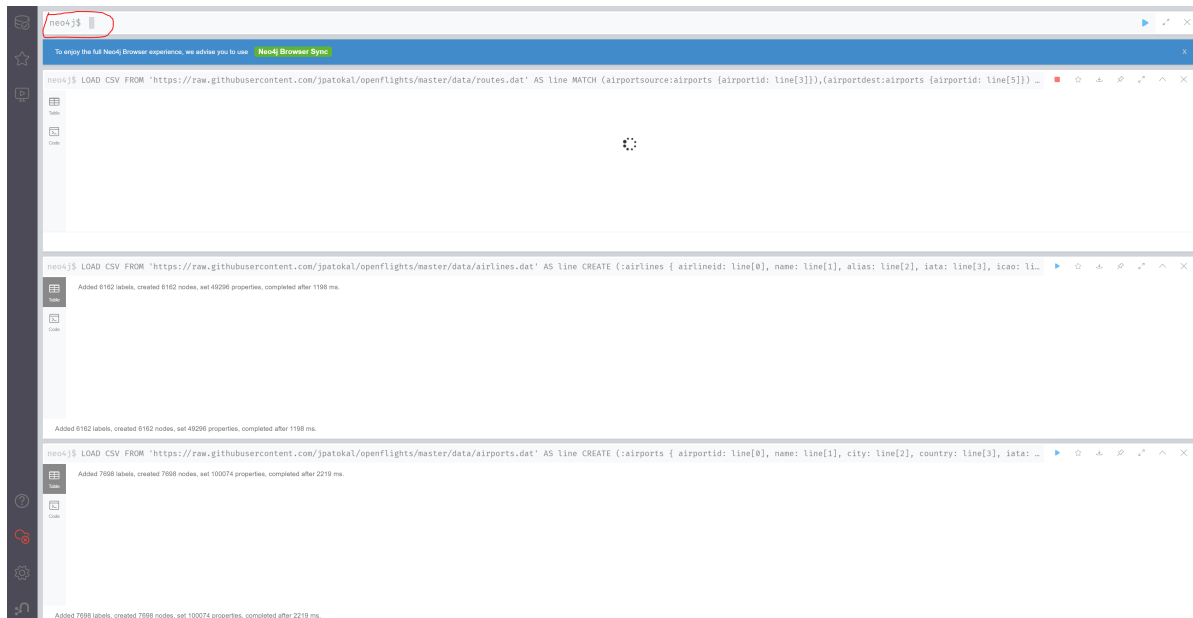
```
# Selecting the address with the maximum number of amenities
{
  '$sort': {
    'n_amenities_per_address': -1
  }
}, {
  '$limit': 1
}
]
```

Wenn man erst einmal die Anzahl an `amenities` pro Adresse berechnet hat (siehe Teilaufgabe 2), dann ist es einfach die Adresse mit den meisten `amenities` zu berechnen. Aufbauend auf dem 1. Lösungsweg von Teilaufgabe 2 habe ich eine `sort` und eine `limit` Stage analog zu Teilaufgabe 1 hinzugefügt.

# Neo4J

Man soll Neo4J und die OpenFlight Datenbank installieren und dann die kürzeste Verbindungen zwischen Berlin und Rio de Janeiro ermitteln.

Die Installation von Neo4J erfolgt auch sehr einfach über Docker, wie man ab Zeile 272 in der [docker-compose](#) Datei sehen kann. Das web-basierte UI von Neo4J ist dann über <http://localhost:7474> erreichbar, siehe Screenshot:



In rot markiert sieht man die Query Konsole, in der man Abfragen mit Cypher, der Abfragesprache von Neo4J, eingeben kann.

Die Daten der OpenFlight Datenbank können mittels folgender Cypher Abfragen eingelesen werden. Zuerst werden die Daten der `airports.dat` Datei eingelesen mit:

```
LOAD CSV FROM
'https://raw.githubusercontent.com/jpatokal/openflights/master/data/airports.dat'
AS line
CREATE (:airports { airportid: line[0], name: line[1], city: line[2], country:
line[3], iata: line[4], icao: line[5], latitude: line[6], longitude: line[7],
altitude: line[8], timezone: line[9], dst: line[10], timezone: line[11], tpe:
line[12], source: line[13]})
```

`airlines.dat` wird eingelesen mit:

```
LOAD CSV FROM
'https://raw.githubusercontent.com/jpatokal/openflights/master/data/airlines.dat'
AS line
CREATE (:airlines { airlineid: line[0], name: line[1], alias: line[2], iata:
line[3], icao: line[4], callsign: line[5], country: line[6], active: line[7]})
```

`routes.dat` wird eingelesen mit:

```

LOAD CSV FROM
'https://raw.githubusercontent.com/jpatokal/openflights/master/data/routes.dat'
AS line
MATCH (airportsource:airports {airportid: line[3]}),(airportdest:airports
{airportid: line[5]})
CREATE (airportsource)-[:R {airlineid:line[1], codeshare: line[6], stops:
line[7], equipment: line[8]}]->(airportdest)

```

und `planes.dat` wird eingelesen mit:

```

LOAD CSV FROM
'https://raw.githubusercontent.com/jpatokal/openflights/master/data/planes.dat'
AS line
CREATE (:planes { iata: line[0],icao: line[1]})

```

Die kürzeste Verbindung zwischen Rio De Janeiro und Berlin wird schließlich folgendermaßen ermittelt:

```

MATCH p=shortestPath((airportsource:airports{city: "Berlin"})-[*]->
(airportdest:airports {city: "Rio De Janeiro"}))
RETURN p

```

Die `shortestPath` Funktion ermittelt in diesem Fall die kürzeste Anzahl an zu überspringenden Relationships, um von Nodes, deren `city` Property `Berlin` ist, zu Nodes, deren `city` Property `Rio De Janeiro` ist, zu gelangen. Da die Relationships aus der Datei `routes.dat` stammen, die non-Stop Verbindungen von einem Flughafen zu einem anderen Flughafen beinhaltet, zeigt das Ergebnis der `shortestPath` Funktion also wie man seine Reise von Berlin nach Rio De Janeiro planen müsste um möglichst wenig umzusteigen.

Das Ergebnis dieser Abfrage sieht dann wie folgt aus:

