



Vorlesung

# Cloud- und Web- Technologien

(CS3140)

## Datenverarbeitung mit Spark und Flink

Professor Dr. rer. nat. habil. Sven Groppe

<https://www.ifis.uni-luebeck.de/index.php?id=groppe>

# Chronologische Übersicht über die Themen

## Nr Thema

1 Einleitung

2 Einführung in das Semantic Web, RDF und SPARQL



Datenmodell

3 Die Semantic Web-Ontologiesprachen RDFS und OWL

4 Multiplattform-Entwicklung mit Kotlin



Multiplattform

5 Fortgeschrittene Themen mit Kotlin

6 Einstieg in Cloud Computing, Hadoop



Backend

7 Operatoren der relationalen Algebra in Hadoop

8 Datenverarbeitung mit Pig

**9 Einführung in Spark und Flink**

10 Stromverarbeitung mit Flink

11 Knotenzentrische Algorithmen mit Flink

12 HTML und CSS

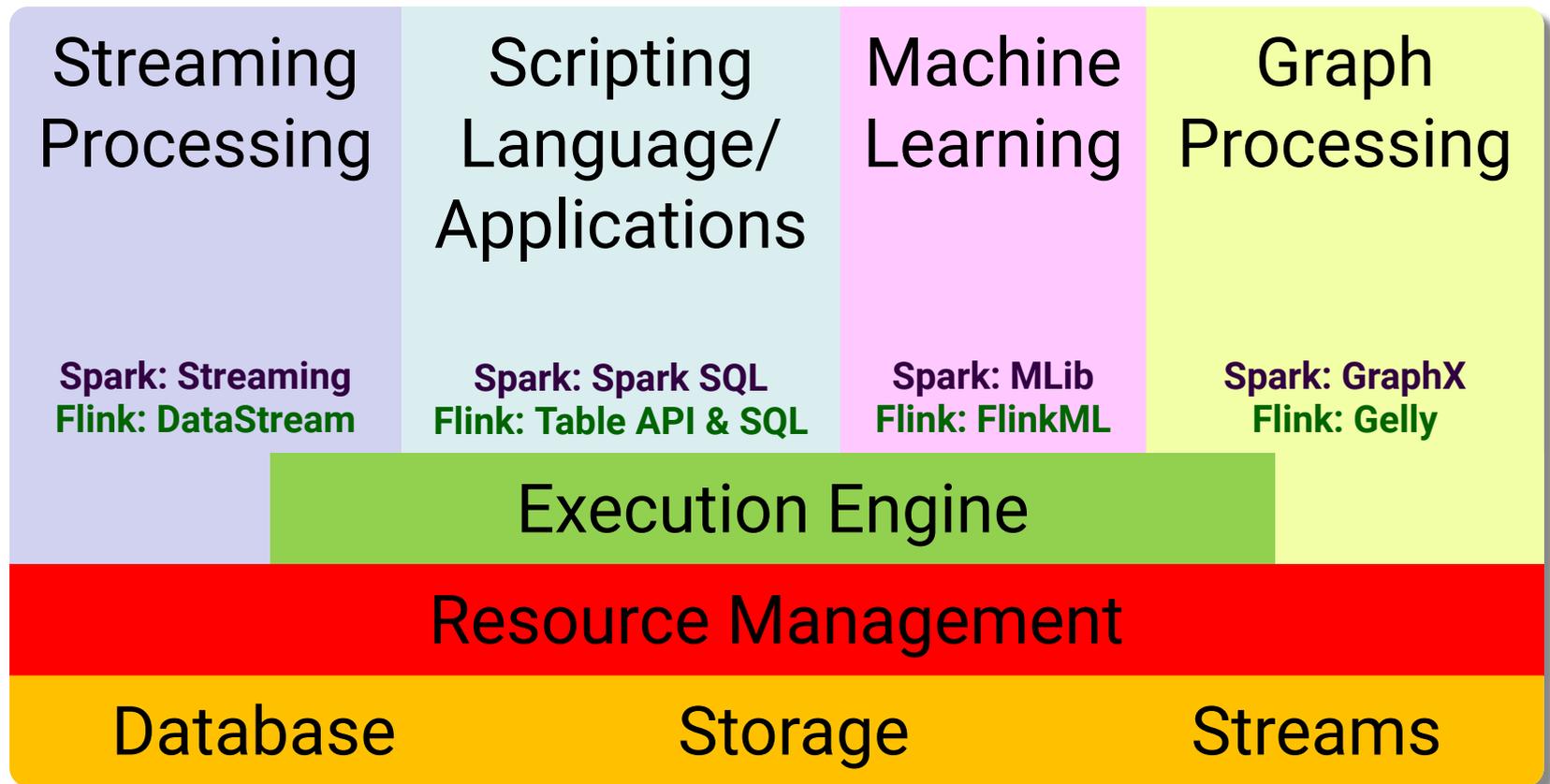
13 Browserprogrammierung mit JS/JQuery und  
Serverprogrammierung mit PHP Hypertext Preprocessor



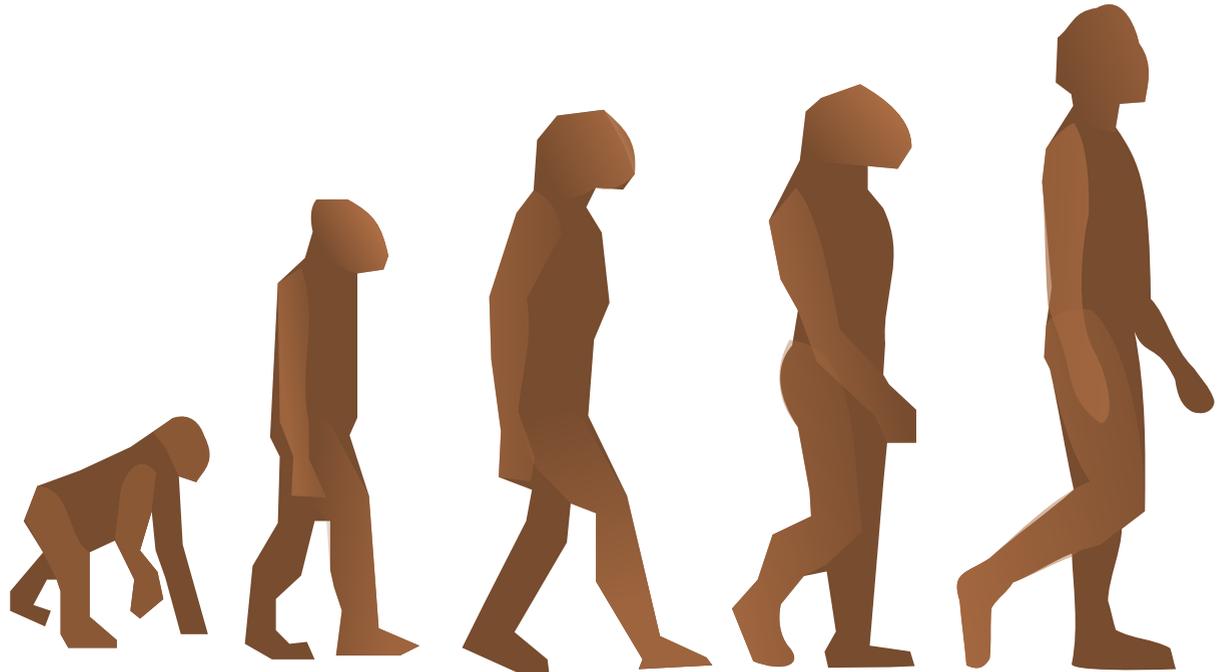
Web

14 Zusammenfassung und Ausblick

# Typischer Big Data Analytics Stack (z.B. Spark, Flink, Storm)



# Evolution der Big Data Analytics Engines



	1. Generation	2. Generation	3. Generation	4. Generation	5. Generation
Features	Batch	+ Interactive	+ Near-Real-Time <sup>1</sup> + Iterative Processing	+ Real-Time Streaming + Native It. Processing	?
Verarbeitungsmodell	MapReduce	DAG Dataflows	Resilient <sup>2</sup> Distributed Datasets (RDD)	Cyclic Dataflows	?
Engine	Hadoop	TEZ	Spark	Flink	?

# Java 8 Stream API - schrittweise



- Parallele Ausführung im Hauptspeicher

```
(1) Path path = Paths.get("C:\\test.txt");
    Map<String, Long> c = Files
(2)     .lines(path) // read all lines from file as stream
        // split lines to words
(3)     .flatMap(line -> Arrays.stream(line.split("\\W+")))
        // for each word generate (word, 1)-tuple
(4)     .map(word -> new SimpleEntry<>(word,1))
        // group be the words and sum it up
(5)+(6)     .collect(Collectors.groupingBy(SimpleEntry::getKey, Collectors.counting()));
    // print out result
    c.forEach( (k,v) -> System.out.println(String.format("%s => %d", k, v)) );
```

(1)

test.txt

Hallo Student

Hallo Dozent

# Spark



# Flink

## • Core

```
SparkSession sc = SparkSession.builder()
    .appName("WordCount").getOrCreate();
// read file
JavaRDD<String> text = sc.textFile("hdfs://...");
JavaPairRDD<String, Integer> counts = text
    // split into words
    .flatMap(s -> Arrays.asList(s.split(" ")).iterator())
    // for each word generate tuple (word, 1)
    .mapToPair(word -> new Tuple2<>(word, 1))
    // group by key and sum up tuple field "1"
    .reduceByKey((a, b) -> a + b);
```

## • Streaming

```
JavaStreamingContext jssc = ...
...
JavaDStream<String> counts = text.flatMap(...)
jssc.start();
jssc.awaitTermination();
```

## • DataSet API

```
ExecutionEnvironment env = ExecutionEnvironment
    .createLocalEnvironment();
DataSet<String> text = env.readTextFile("c:\\...");
DataSet<Tuple2<String, Integer>> counts = text
    // split up lines in words
    .flatMap((FlatMapFunction<String, String>)(s, o)
        -> {for(final String r: s.toLowerCase()
            .split("\\W+")){o.collect(r);}})
    .returns(String.class)
    // generate tuples (word, 1)
    .flatMap((FlatMapFunction<String,
        Tuple2<String, Integer>>)(s, o)
        -> {o.collect(new Tuple2<String, Integer>(s,1));})
    .returns(new TypeHint<Tuple2<String, Integer>>(){}))
    .groupBy(0) // group by the tuple field "0"...
    .sum(1); // ...and sum up tuple field "1"
```

## • DataStream API

```
StreamExecutionEnvironment env =
    StreamExecutionEnvironment.getExecutionEnvironment();
DataStream<String> text = env.readTextFile("c:\\...");
DataStream<Tuple2<String, Integer>> counts = text
    ... .keyBy(value -> value.f0).sum(1);
env.execute("Streaming Example");
```

# Beobachtungen

- Ähnliche APIs
  - Java 8, Spark, Flink
  - Batch-Verarbeitung, Streaming bei Spark/Flink
- Unterschiede
  - Initialisierung der Ausführungsumgebung
  - Unterschiedliche Klassen für die Repräsentation der Datensätze/-ströme
    - Mit vielen gleichnamigen Methoden  
( $\rightsquigarrow$  ähnliche Verwendung der APIs)
    - Fehlende und zusätzliche Methoden teilweise mit ähnlicher, aber nicht gleicher Bedeutung
    - Ausgabe
      - Flink DataSet API: (dozent,1) (student,1) (hallo,2)
      - Flink DataStream API: (student,1) (hallo,1) (hallo,2) (dozent,1)  
*Kontinuierliche und iterative Stromverarbeitung,  
„rollende“ Ergebnisse...*

# Spark



- Unterstützung von Batch- und Stromverarbeitung
- Sowohl API auf Tabellen als auch ganze SQL-Anfragen

- Kompatibel zu Hive
  - Daten, Anfragen, UDFs

```
df.filter(col.get("age").gt(21))  
  .select(col("name"));
```



```
df.sql("SELECT name FROM table where age>21");
```

- Im Vergleich dazu relativ eingeschränktes SQL

```
table.where("age > 21")  
  .select(col("name"));
```



```
table.sqlQuery("SELECT name FROM table where age>21");
```

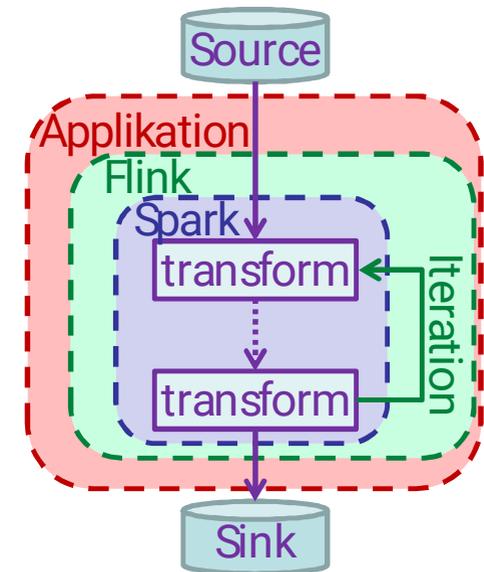
## Vorteile von Flink 1/2

- **Nativ in Java** programmiert und dadurch stimmigere Java-API  
(Spark ist in **nativ Scala** programmiert)
- **Besseres Memory-Management**
  - **Serialisierung von Daten** in Byte-Arrays,
  - **Buffermanagement vergleichbar zu DBMS**
  - **eigene Speicherverwaltung**  
(anstelle von zeitraubender Garbage Collection)  
vergleichbar zu C/C++

## Vorteile von Flink 2/2

- **Automatische Optimierung**  
(z.B. Filter-Push-Down) der Verarbeitung  
(in Spark oft **manuelle Optimierung** notwendig)

- **Unterstützung & Optimierung von Iterationen**  
(`.iterate(...)` / `.iterateDelta(...)`)
  - Iterationen in Spark (und z.B. auch in Hadoop) in **Applikationslogik**, dadurch Overhead (z.B. Initialisierungen) in jeder Schleifeniteration



- **Aber: Spark holt** technologisch **auf** und entwickelt aktuelle APIs weiter (z.B. Memory-Management)...

# Apache Flink API – Anatomie eines Flink-Programms 1/2

## 1. Beziehe eine **Ausführungsumgebung**

- Basisklasse `ExecutionEnvironment` oder `StreamExecutionEnvironment` für Streams
  - `createLocalEnvironment()`: lokale Ausführungsumgebung
  - `createRemoteEnvironment(String host, int port, String... jarFiles)`: Cluster-Ausführungsumgebung
  - `getExecutionEnvironment()`: je nach Kontext Cluster- oder lokale Ausführungsumgebung

## 2. **Lade/kreiere** die initialen **Daten**

- Durch Angabe einer Quelle, Standardquellen in Methoden der Ausführungseinheit

## 3. Spezifiziere **Transformationen** auf diesen Daten

# Apache Flink API – Anatomie eines Flink-Programms 2/2

## 4. **Spezifiziere den Ablageort** der Ergebnisse

- Durch Angabe einer Senke (engl. sink),  
z.B. `print()`, `writeAsText(String path)`

## 5. **Trigger** die Programmausführung

- Batch-Verarbeitung: bereits durch Angabe einer Senke beginnt Ausführung
- Stream-Verarbeitung: `environment.execute("Stream-Job");`

Viele der Ausführungen gelten analog für Spark (bis auf konkrete API-Calls und vorher erwähnte Unterschiede)

# Apache Flink – Lazy Evaluation

- „initiale Daten laden/kreieren“ und „transformieren“
  - Hinzufügen von Operatoren zum Programmplan
  - Ausführung der Operatoren passiert nicht augenblicklich, sondern erst nach dem Triggern (und Optimierung des gesamten Programmplans)
  - Berechnung von nur so vielen Zwischenergebnissen wie notwendig für das Gesamtergebnis
    - Bsp.: `env.readCsvFile(...).flatMap(...).first(2).print();`  
liest und verarbeitet nur **zwei Zeilen** des CSV-Files  
(vgl. Iteratorkonzept bei Datenbanken)

# Apache Flink – Unterstützte Datentypen 1/2

- **Primitive Typen** (z.B. Integer, String, Double)
- **Java Tupel und Scala Case Classes**
  - Zusammengesetzte Datentypen
  - Tuple1 bis Tuple25, z.B. `new Tuple2<String, Integer>("hello", 1);`
- **Java Plain old Java Objects (POJOs)**
  - Klasse muss `public` sein mit einem `public` Konstruktor ohne Argumente (Default-Konstruktor)
  - Alle Felder entweder `public` oder durch Getter- und Setter-Methoden erreichbar  
Namensschema:  
`name`  $\Rightarrow$  `setName(...)`, `getName()`
  - Unterstützung der Feld-Datentypen durch Flink
    - Zurzeit verwendet Flink Avro zur (De-)Serialisierung

```
public class WC {  
    public String word;  
    private int count;  
    public void setCount(int c){  
        this.count = c;  
    }  
    public int getCount(){  
        return this.count;  
    }  
}
```

# Apache Flink – Unterstützte Datentypen 2/2

- **Reguläre Klassen**

- (De-)Serialisiert durch Kryo
  - keine Unterstützung von File Pointers, I/O Streams oder andere native Ressourcen
- Für Flink „Black Boxes“, teils **kein effizienter Zugriff**

- **Values**

- Implementation der Value-Schnittstelle
  - `read(...)` und `write(...)`-Methoden zur expliziten Angabe der (De-)Serialisierung
- Einige vordefiniert durch Flink:  
ByteValue, ..., StringValue, BooleanValue

- **Hadoop Writables**

- **Spezielle Typen**

- z.B. von Scala: Either, Option, Try

# Apache Flink API – Schlüssel

- Angabe eines Schlüssels notwendig für
  - join, coGroup, keyBy, groupBy bei Methodenaufwurf
  - Reduce, GroupReduce, Aggregate, Windows: Anwendung auf vorher (nach einem Schlüssel) gruppierten Daten
- **Schlüssel** nur „virtuell“, keine (!) Speicherung in Key-Value-Stores o.ä.
- **Angabe des Schlüssels durch**
  - **Position**: `input.keyBy(0)`, `input.keyBy(0, 1)`
  - **Feldausdruck**: `words.keyBy("word")`, `words.keyBy("wc.wc.word")`
  - **Key Selektor-Funktion**:

```
words.keyBy(new KeySelector<WC, Integer>() {  
    public Integer getKey(WC wc) {  
        return wc.word.length();  
    }});
```

```
public class WC {  
    public String word;  
    public WC wc;  
}
```

# Apache Flink API – “Type Erasure”

- Java Compiler wirft nach dem Kompilieren viele Typinformationen generischer Klassen weg  
↳ **Type Erasure**
  - Bsp.: Für die JVM nicht unterscheidbar:  
`DataStream<String>` und `DataStream<Long>`
- Aber: Flink benötigt zur Laufzeit Typinformationen (z.B. zum Deserialisieren)

Verwendung des Eclipse JDT Compiler enthalten in Eclipse Luna 4.4.2 (und höher) empfohlen, da dieser weniger Typinformationen wegwirft...

# Apache Flink API – Typinferenz

- **Vielfältige automatische Typinferenz** durch Flink
  - `DataStream.getType()` liefert Instanz von `TypeInformation` zurück (Flinks Typrepräsentation)
- **Manuelle Typangabe** (bei Grenzen der automatischen Inferenz  $\rightsquigarrow$  **Laufzeitfehler**) durch
  - `.returns(...)`-Methoden
  - **Extra-Parameter** bei manchen Methoden (z.B. `.fromCollection(...)`)
  - **Implementierung der `ResultTypeQueryable`-Schnittstelle** bei Eingabeformaten und Funktionen

# Apache Flink API – Spezifikation von Transformationsfunktionen

## Implementation einer Schnittstelle

```
class MyMapFunction implements MapFunction<String, Integer> {  
    public Integer map(String value) { return Integer.parseInt(value); }  
};  
data.map(new MyMapFunction());
```

## + Anonyme Klassen

```
data.map(new MapFunction<String, Integer> () {  
    public Integer map(String value) { return Integer.parseInt(value); }  
});
```

## + Java 8 Lambdas

```
data.filter(s -> s.startsWith("http://"));
```

## Rich Functions

```
class MyMapFunction extends RichMapFunction<String, Integer> {  
    public Integer map(String value) { return Integer.parseInt(value); }  
};
```

- Bereitstellung zusätzlicher Methoden:  
open, close, getRuntimeContext, setRuntimeContext
- Dadurch Parametrisierung der Funktionen, Initialisierungs- und Finalisierungscode, Zugriff auf Broadcast-Variablen und Laufzeitinformationen möglich

# Apache Flink API – Wichtige gemeinsame Methoden von DataSet und DataStream

map

- Bildet genau ein Element auf ein anderes ab

```
data.map(new MapFunction<String, Integer>() {  
    public Integer map(String value) { return Integer.parseInt(value); }  
});
```

flatMap

- Bildet ein Element auf beliebig viele (inklusive 0) Elemente ab

```
data.flatMap(new FlatMapFunction<String, String>() {  
    public void flatMap(String value, Collector<String> out) {  
        for (String s : value.split(" ")) { out.collect(s); }  
    }  
});
```

reduce

- Kombiniert eine Gruppe von Elementen in ein einzelnes durch Reduktion. reduce kann auf den ganzen Datensatz oder eine Gruppe angewendet werden.
- Bei DataStream wird pro verarbeitetes Element ein Zwischenergebnis weitergegeben („Rollende“ Berechnung)

```
data.reduce(new ReduceFunction<Integer> {  
    public Integer reduce(Integer a, Integer b) { return a + b; }  
});
```

# Apache Flink API – Wichtige gemeinsame Methoden von DataSet und DataStream

## aggregate

- Aggregation von vielen Elementen in ein einzelnes als vordefinierte reduce-Funktionen

```
DataSet<Tuple3<Integer, String, Double>> output = input.aggregate(SUM, 0).and(MIN, 2);
```

- Short-Hand Syntax:

```
DataSet<Tuple3<Integer, String, Double>> output = input.sum(0).andMin(2);
```

## filter

- Evaluiert eine *Wahrheitsfunktion* und behält nur die Elemente, für die die Wahrheitsfunktion *wahr* ist

```
data.filter(new FilterFunction<Integer>() {  
    public boolean filter(Integer value) { return value > 1000; }  
});
```

## distinct

- **Nur DataSet:** Eliminierung von Duplikaten

```
data.distinct();
```

## project

- **Bei Verarbeitung von Tupeln (Tuple1 bis Tuple25) zusätzlich:** Projizieren auf eine Teilmenge der Felder

```
DataSet<Tuple2<String, Integer>> out = in.project(2,0);
```

# Apache Flink API – Wichtige gemeinsame Methoden von DataSet und DataStream

union

- Vereinigung zweier DataSets/DataStreams

```
DataSet<String> result = data1.union(data2);
```

- Kombiniert zwei Datensätze bei Gleichheit zweier Felder

```
result = input1.join(input2)
                .where(0)           // key of the first input (tuple field 0)
                .equalTo(1);       // key of the second input (tuple field 1)
```

- **Bei DataStream** Join nur über die Daten eines Fensters (hier Daten der letzten 3 Sekunden (nicht überlappend)):

join

```
dataStream.join(otherStream)
            .where(<key selector>.equalTo(<key selector>)
            .window(TumblingEventTimeWindows.of(Time.seconds(3)))
            .apply(new JoinFunction () {...});
```

- Optional:  
Explizite Angabe einer `JoinFunction` zur Berechnung des Joinergebnisses (z.B. Vereinigung der Felder)

# Apache Flink API – Nur DataStream

window

- Fenster zerteilen den Strom in endlich große Ausschnitte, über die Berechnungen durchgeführt werden können
  - „Keyed Windows“: Zerteilen & parallele Ausführung anhand eines Schlüssels

```
stream.keyBy(...)           // split according to key
      .window(...)           // define the window
      .reduce/fold/apply(...) // determine window-result in parallel
```

- „Non-keyed Windows“: Nur 1 Fenster (in einem Task berechnet)

```
stream
      .windowAll(...)         // define the window
      .reduce/fold/apply(...) // determine window-result in one task
```

# Zum Weiterlesen...

- Apache Spark
  - [Projektwebseite](#)
- Apache Flink
  - [Projektwebseite](#)
  - Apache Flink basiert auf Stratosphere:  
Alexandrov et al., **The Stratosphere platform for big data analytics**. The VLDB Journal 23 (6), 2014.  
[DOI: 10.1007/s00778-014-0357-y](https://doi.org/10.1007/s00778-014-0357-y)

# Zusammenfassung

- Apache Spark/Flink als 3./4. Generation von Cloud Computing-Frameworks
  - Umfangreiche APIs, z.B.
    - Batch-Verarbeitung
    - Stromverarbeitung
    - SQL
- Apache Flink zurzeit teilweise bessere Backendtechnologien