Big-Data-Technologien

Kapitel 7: Spark

Hochschule Trier
Prof. Dr. Christoph Schmitz

Überblick

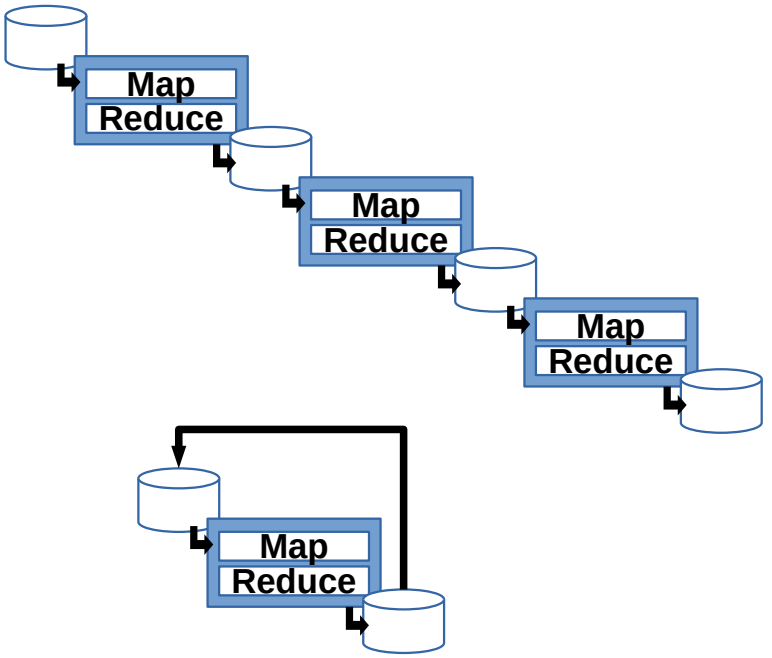
- Warum Spark?
- Datenstruktur RDD
- Funktionale Konzepte in Spark
- Schmale und breite Transformationen
- Spark Streaming
- Data Frames
- Spark SQL

Was ist Spark?

- Framework für **verteilte Berechnungen** im Batch-Modus
- Vergleichbar mit **Hadoop MapReduce**
- Wesentlich **jünger** als Hadoop
- **Reichhaltigere Operationen**
- **Effizientere** Berechnungen möglich
- Erweiterungen für **Streaming, SQL, Graphen,** …

Warum Spark?

- **MapReduce**
  - Nur zwei Operationen
  - Viele persistierte Zwischenergebnisse
  - Nicht geeignet für iterative Verfahren:




```
x ← x0
data ← read(…)
do
   x ← f(x, data)
while not converged(x)
```

Groß

M/R

„Faktor 100!“

Was bietet Spark?

- **Reichhaltigere Operationen** für Datenflüsse
  - ca. 50 statt nur zwei
  - auch die unangenehmen:
    join, sort, group, countApproxDistinct
- **RDD (Resilient Distributed Dataset)**
  - Abstraktion für verteilte Datensätze
  - Grundlage für verteilte Berechnung
- **Caching** von Daten

- **Ökosystem:**
  - Streams, SQL, Machine Learning, Graphen, …

Hello World

- **WordCount** in Spark (Scala):

```
sc.textFile("something.txt")
  .flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)
  .saveAsTextFile("wordcount.csv")
```

Abstraktion RDD

Resilient Robuster, fehlertoleranter

Distributed verteilter

Dataset Datensatz

Resilient Distributed **Dataset**

- Multimenge von **Objekten**

```
müller
meier
schmidt
becker
meier
```

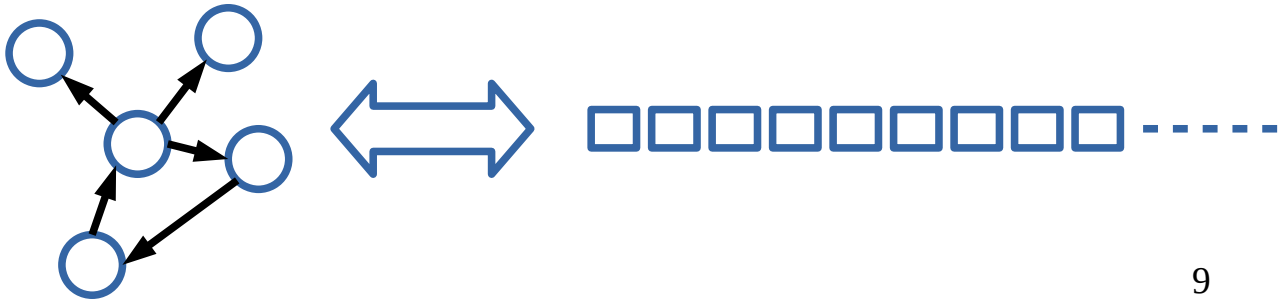
- Multimenge von **Name-Wert-Paaren**

```
(123, müller)
(124, meier)
(123, schmidt)
(125, becker)
(124, meier)
```

Datentypen in RDDs

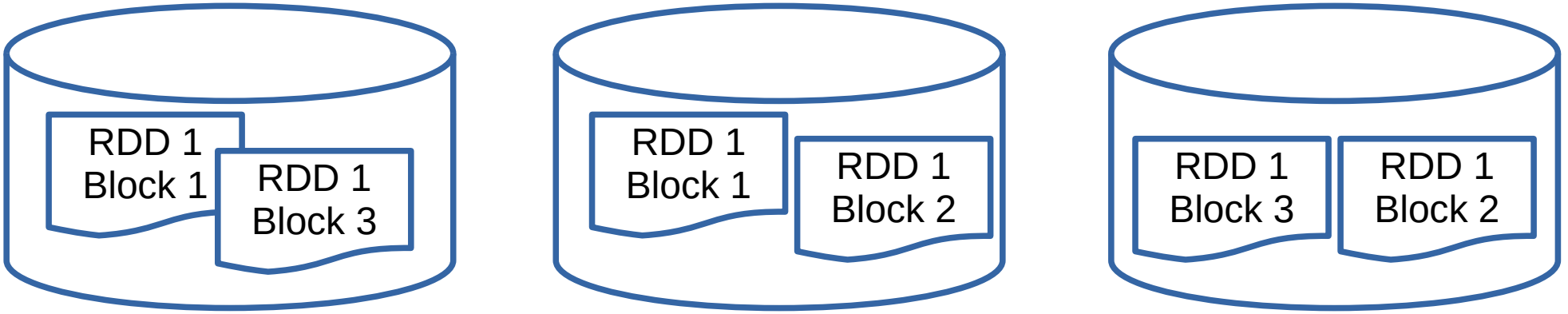
- Daten müssen **serialisierbar** sein
  - **Objekt → Bytes** (Serialisierung, Marshalling)
  - **Bytes → Objekt** (Deserialisierung, Unmarshalling)

- Verschiedene Mechanismen möglich
  - Java **`Serializable`**
  - **Kryo**



RDDs im HDFS

- Resilient **Distributed** Dataset
- Verteilung im **HDFS**



- Verteilung in der **Berechnung**

Resilient Distributed Dataset

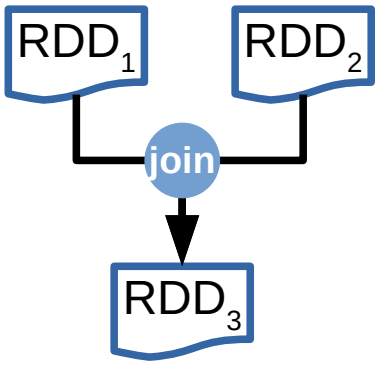
RDD: Zwei Sichten auf Daten

- **Intension (Inhalt)**
  - Beschreibung über Eigenschaften: „Frauen über 50“
  - Berechnungsvorschrift („Herkunft“, „*lineage*“)
- **Extension (Umfang)**
  - Aufzählung der konkreten Datensätze
  - Daten im HDFS

Name	Geschlecht	Alter
Anna	w	52
Barbara	w	65
Cäcilia	w	54
…	…	…

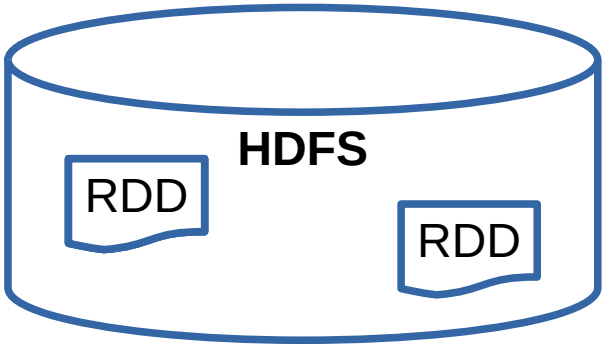
Wo kommen RDDs her?

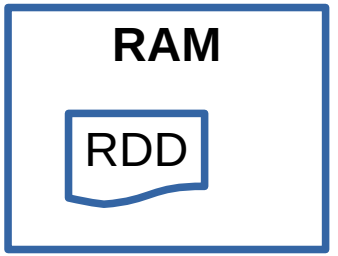
- **Intensional**
  - Berechnen aus anderen RDDs



$RDD_3 = RDD_1.join(RDD_2)$

- **Extensional**
  - Importierte Daten
  - Zwischenergebnisse





Datenbanken: (Materialized) Views

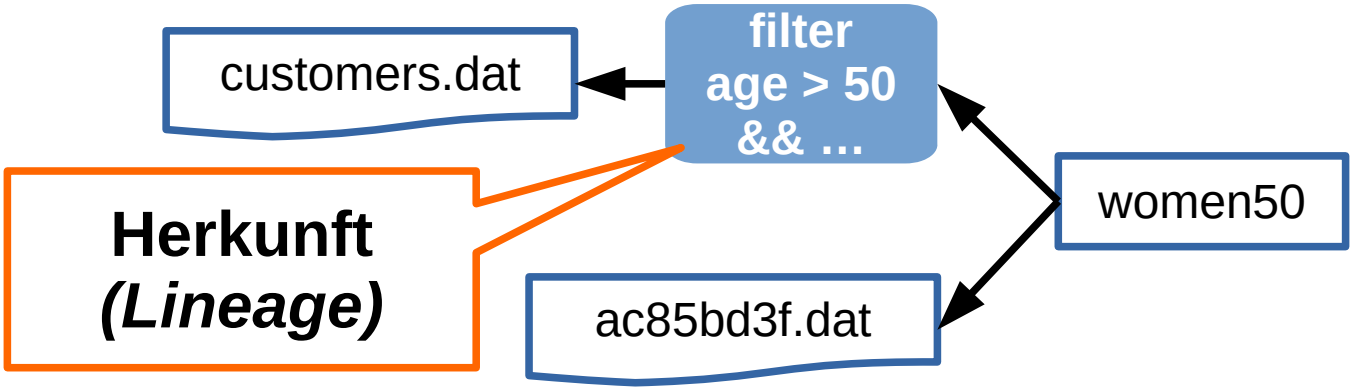
Rückschau: Funktionale Programmierung

- Referenzielle Transparenz:

  d = f(a, b, c) → d und f(…) sind **gleichwertig**!

- RDDs:

```
RDD customers = spark.load("customers.dat");
RDD women50 = customers.filter(
      c → c.age > 50 && c.gender == w);
women50.persist(MEMORY_AND_DISK);
```

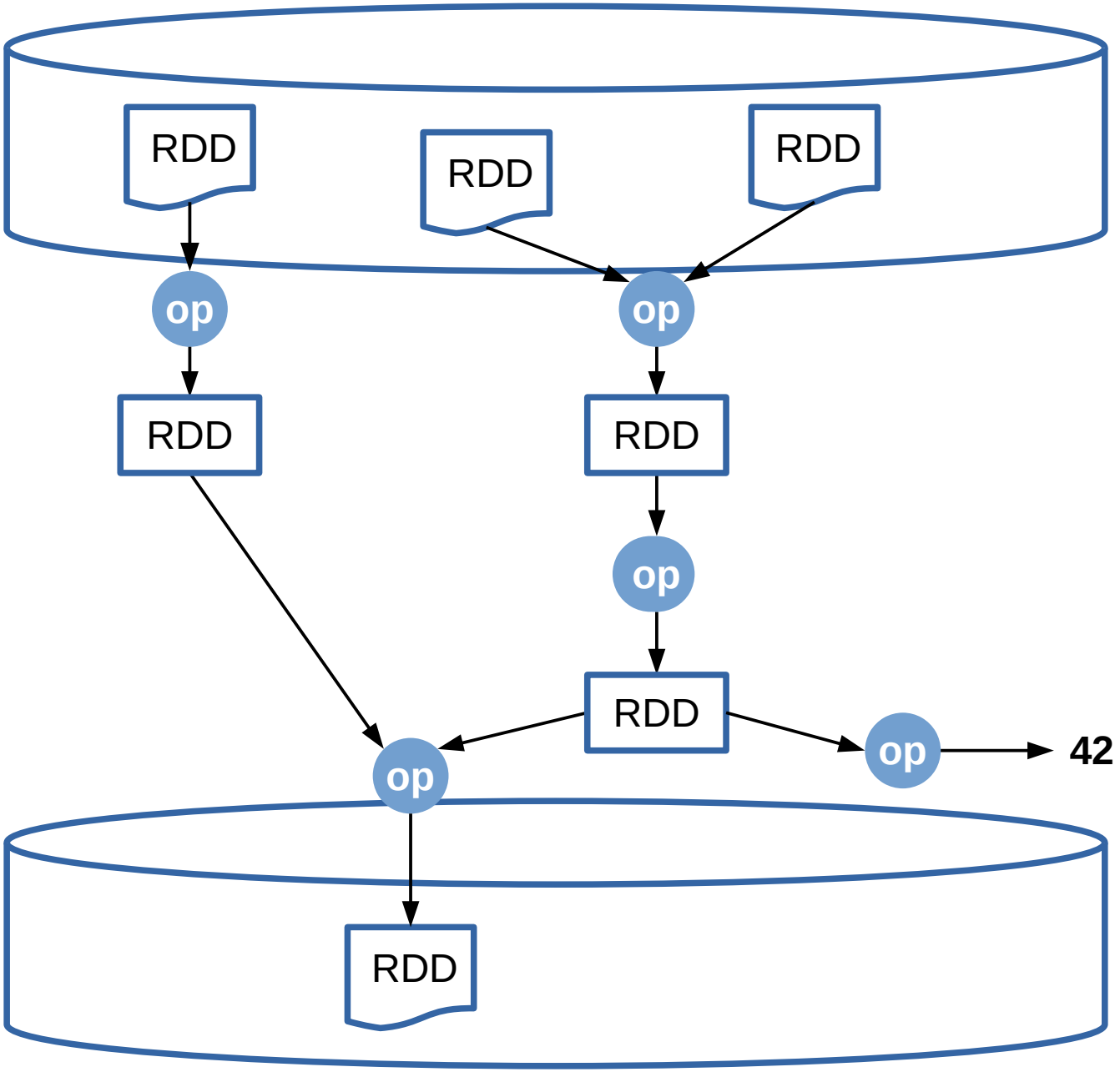


Funktionale Konzepte und RDDs

- **Unveränderlichkeit**
  - RDDs sind immutable
  - Können repliziert werden
- **Referenzielle Transparenz**
  - Extension und Intension austauschbar
  - Neuberechnung möglich
- **Keine Seiteneffekt**
  - Kein verteilter Zustand
- **Lazy Evaluation**
  - Berechnung leicht verteilbar
  - Nur rechnen, wenn es unvermeidlich ist

Resilience

Datenflüsse mit RDDs



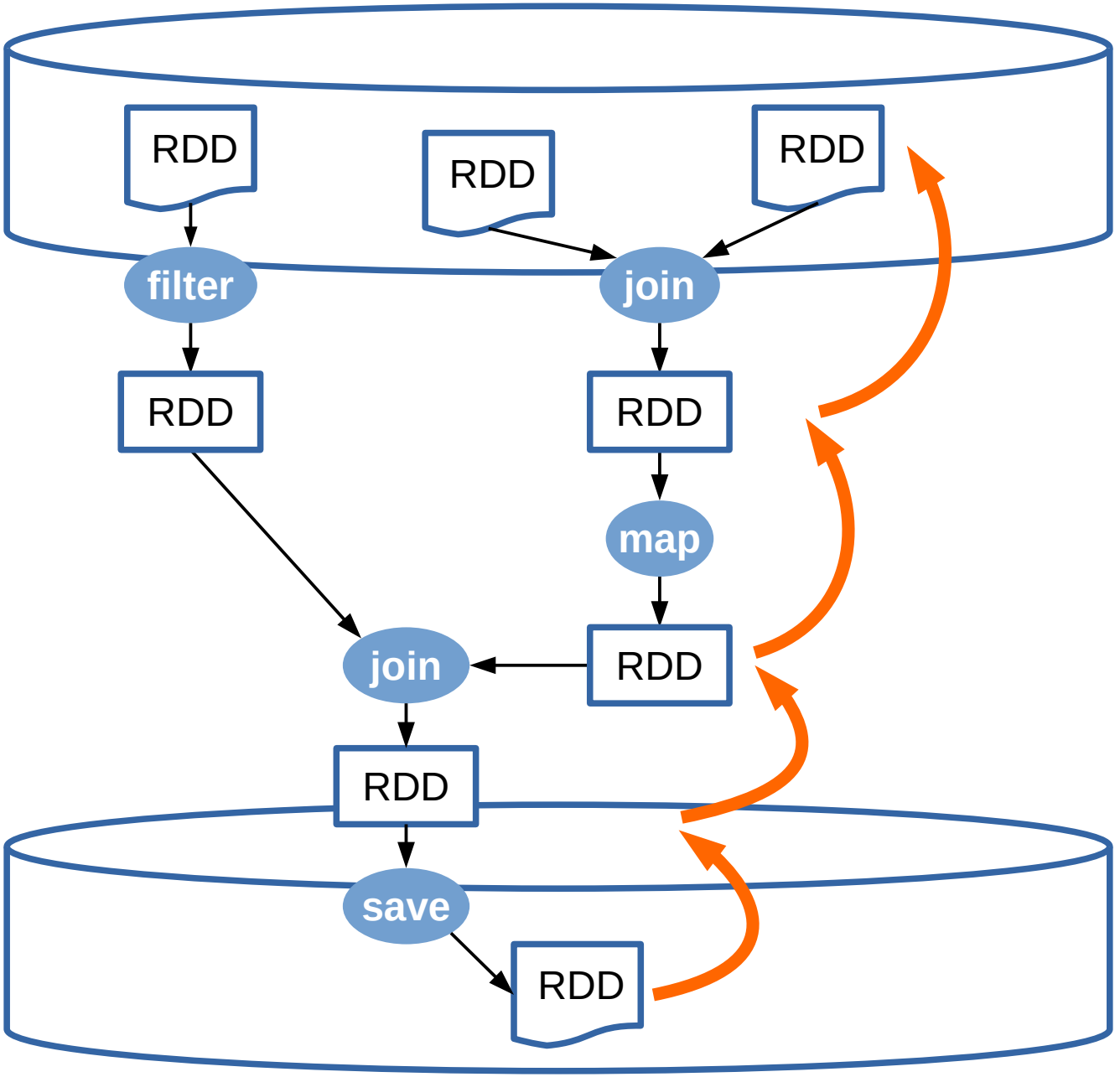
Arten von Operationen

- **Transformationen**
  - verrechnen RDDs zu neuen RDDs
  - **Schmale Transformationen**
    - Verarbeiten Datensätze einzeln (vgl. Map)
  - **Breite Transformationen**
    - Beziehungen zwischen Datensätzen (vgl. Reduce)

Embarrassingly Parallel

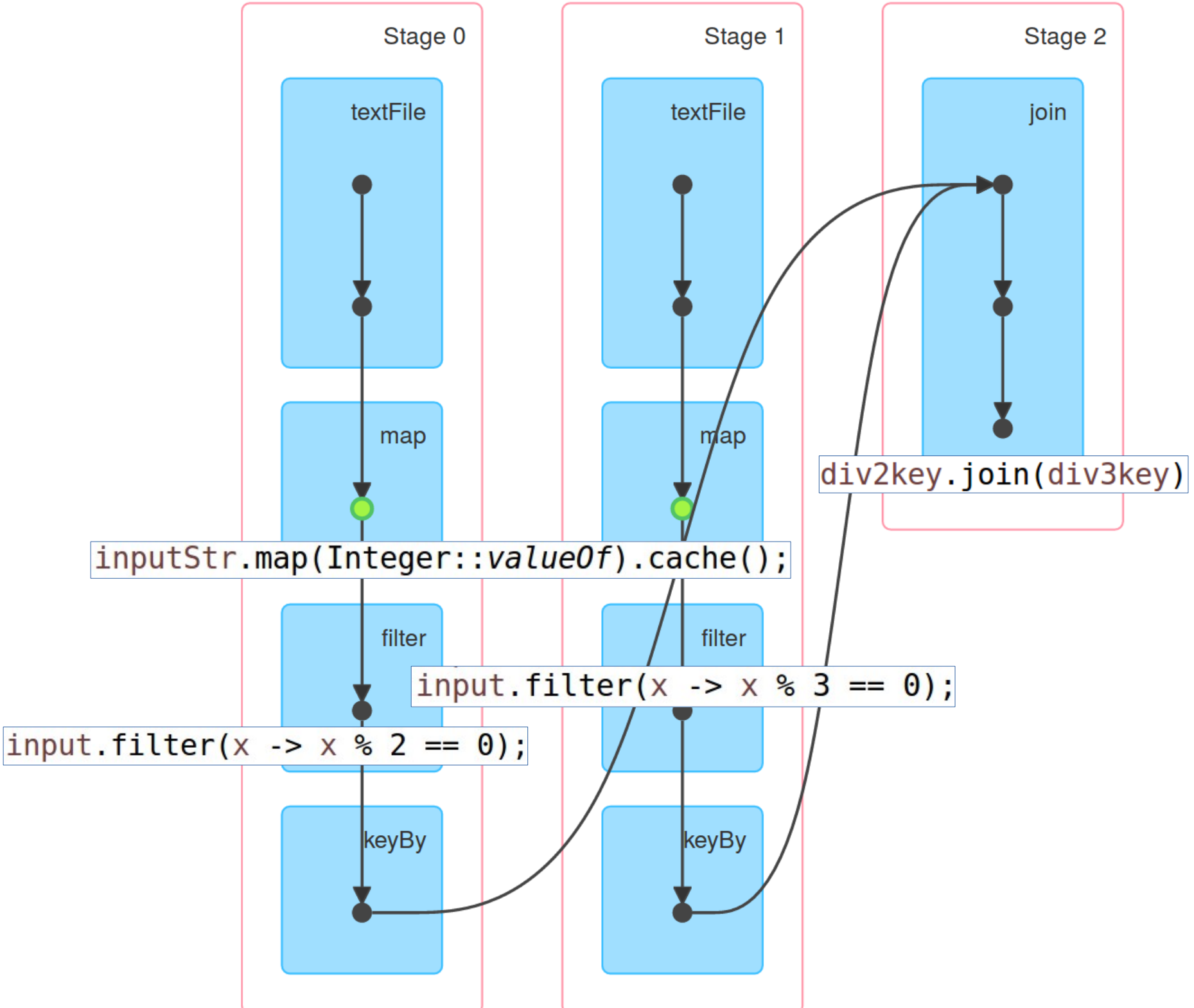
- **Aktionen**
  - Resultat ist kein RDD
    - z. B. `count`, `save`
  - **Erst Aktionen lösen Berechnung aus!**

Datenflüsse und Ausführungspläne

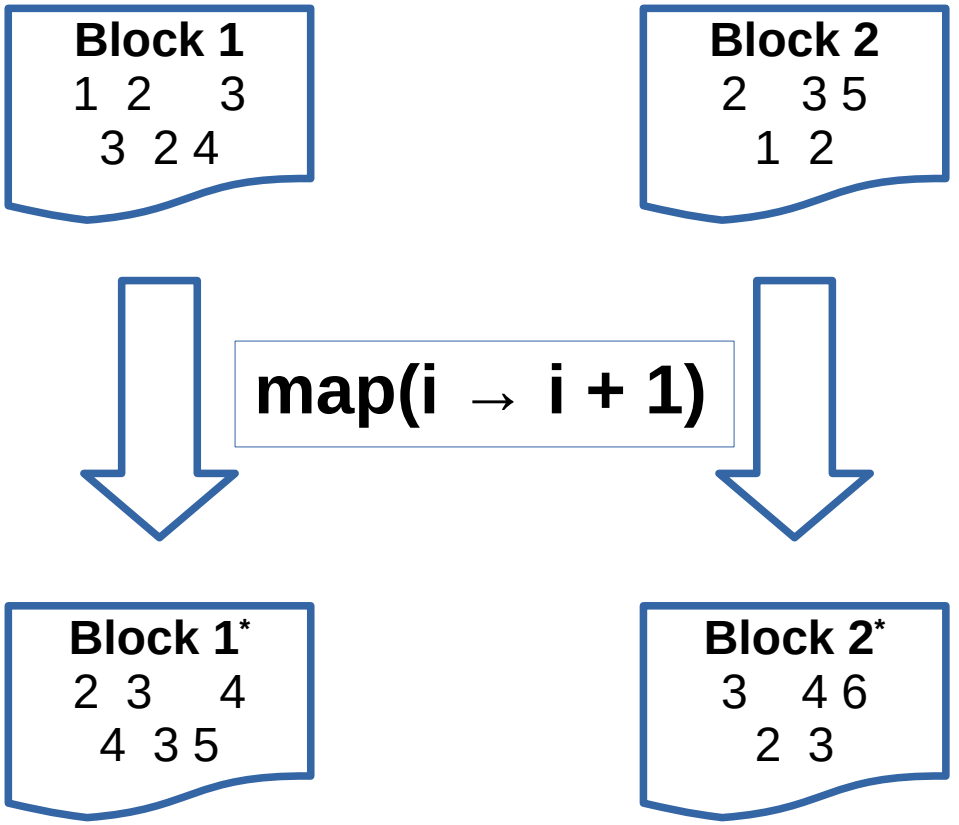


Code!

```
SparkConf conf = new SparkConf().setAppName("rdd-test").setMaster("local");
try (JavaSparkContext ctx = new JavaSparkContext(conf)) {

    JavaRDD<String> inputStr = ctx.textFile("/tmp/data.txt");
    JavaRDD<Integer> input = inputStr.map(Integer::valueOf).cache();

    JavaRDD<Integer> div2 = input.filter(x -> x % 2 == 0);
    JavaRDD<Integer> div3 = input.filter(x -> x % 3 == 0);

    JavaPairRDD<Integer, Integer> div2key = div2.keyBy(i -> i);
    JavaPairRDD<Integer, Integer> div3key = div3.keyBy(i -> i);

    long div6count = div2key.join(div3key).count();

    System.err.println(div6count);
}
```

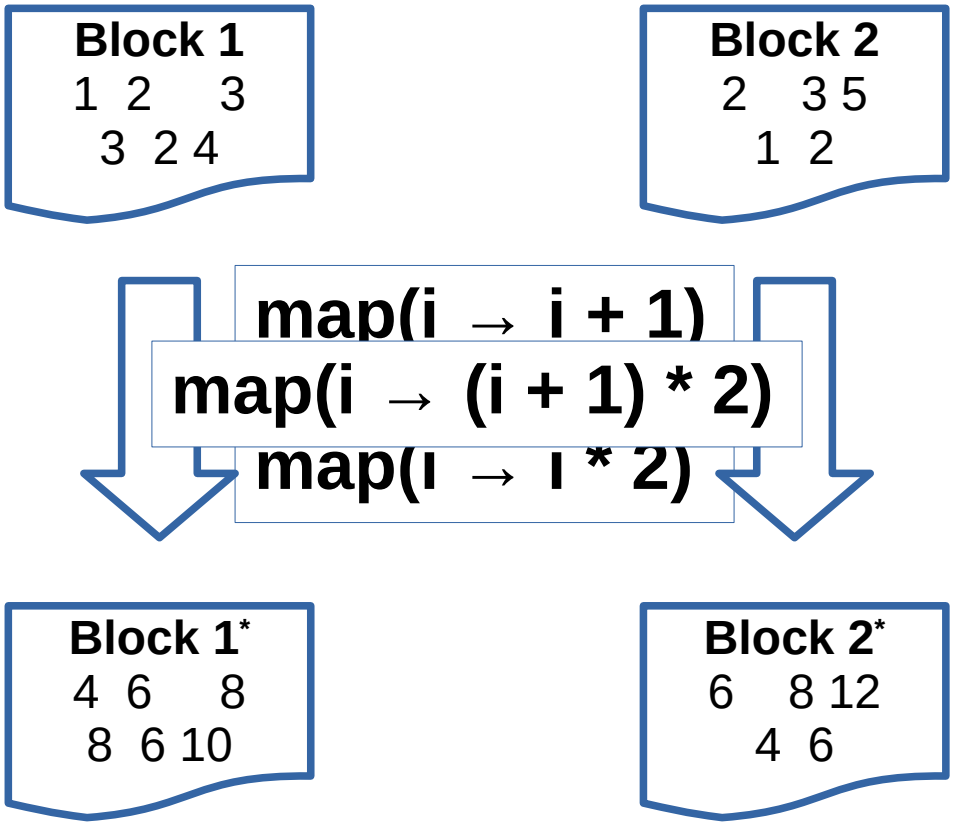

Stage 0
Stage 1
Stage 2
textFile
textFile
join
map
map
filter
filter
keyBy
keyBy
inputStr.map(Integer::valueOf).cache();
input.filter(x -> x % 2 == 0);
input.filter(x -> x % 3 == 0);
div2key.join(div3key)

Ausführen von Operationen: Schmale Transformation



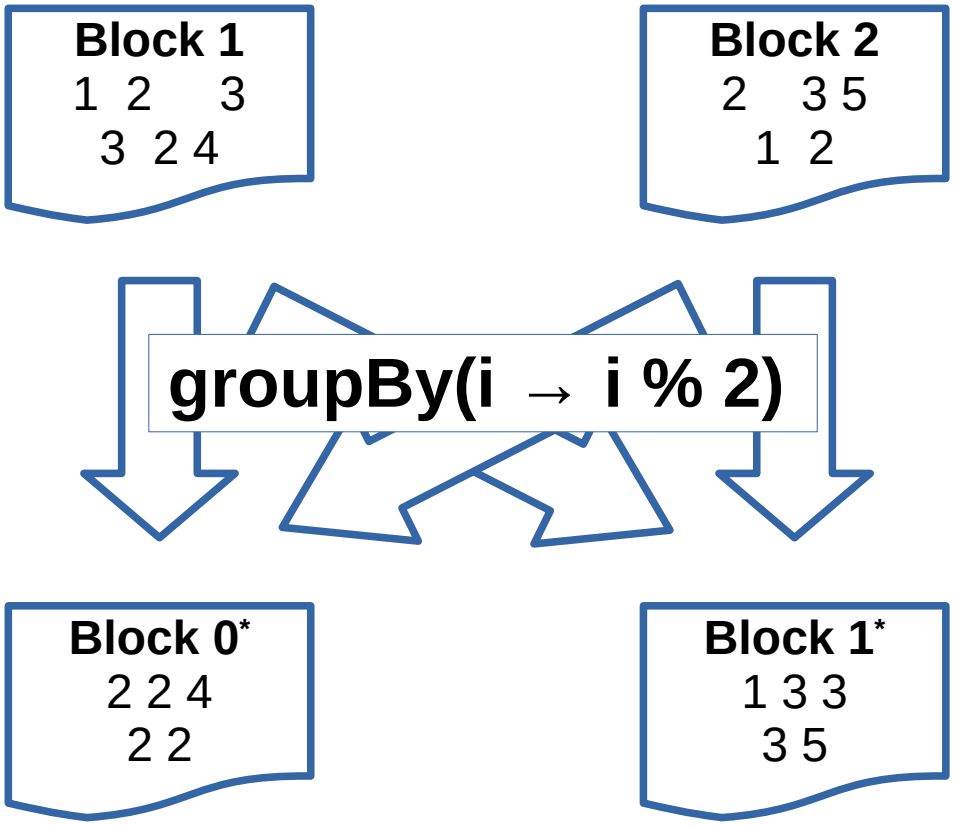
Embarrassingly Parallel

Zusammenfassen schmaler Transformationen



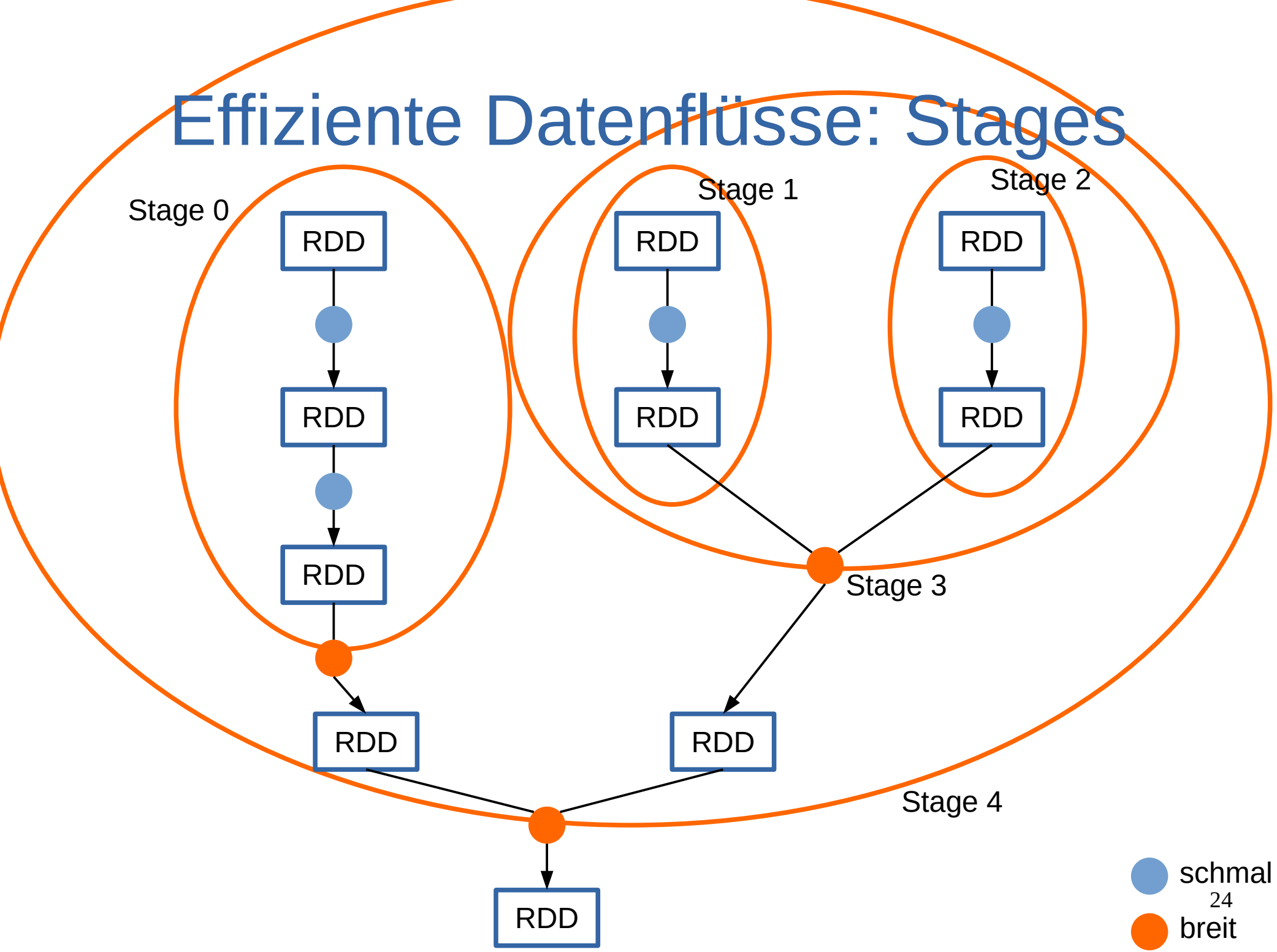
Embarrassingly Parallel

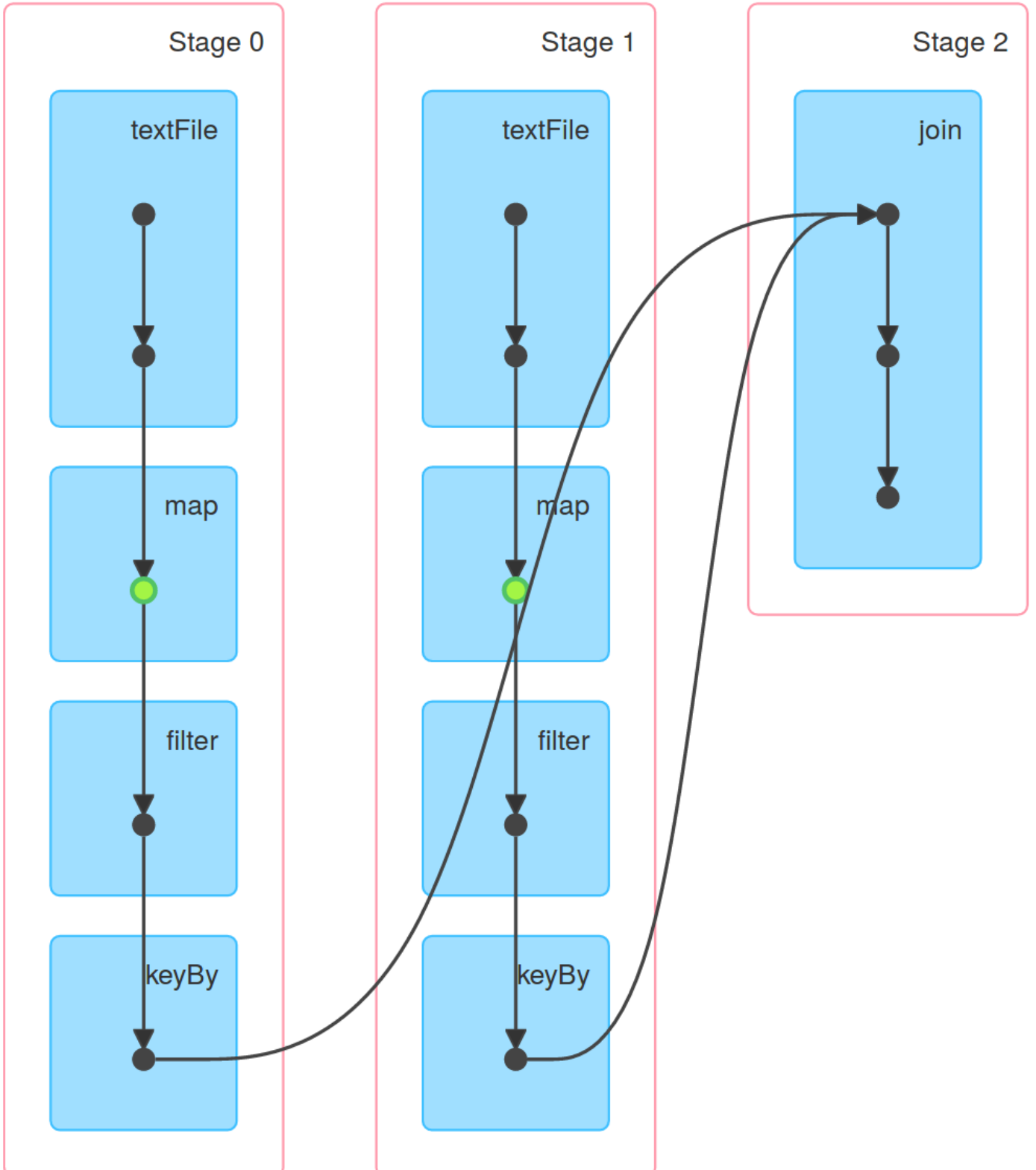
Ausführen von Operationen: Breite Transformation



Shuffle

Effiziente Datenflüsse: Stages




Stage 0
textFile
map
filter
keyBy
Stage 1
textFile
map
filter
keyBy
Stage 2
join

Verfügbare Operationen

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

Quelle: Zaharia et al., „Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing“, 2012

Fluent APIs

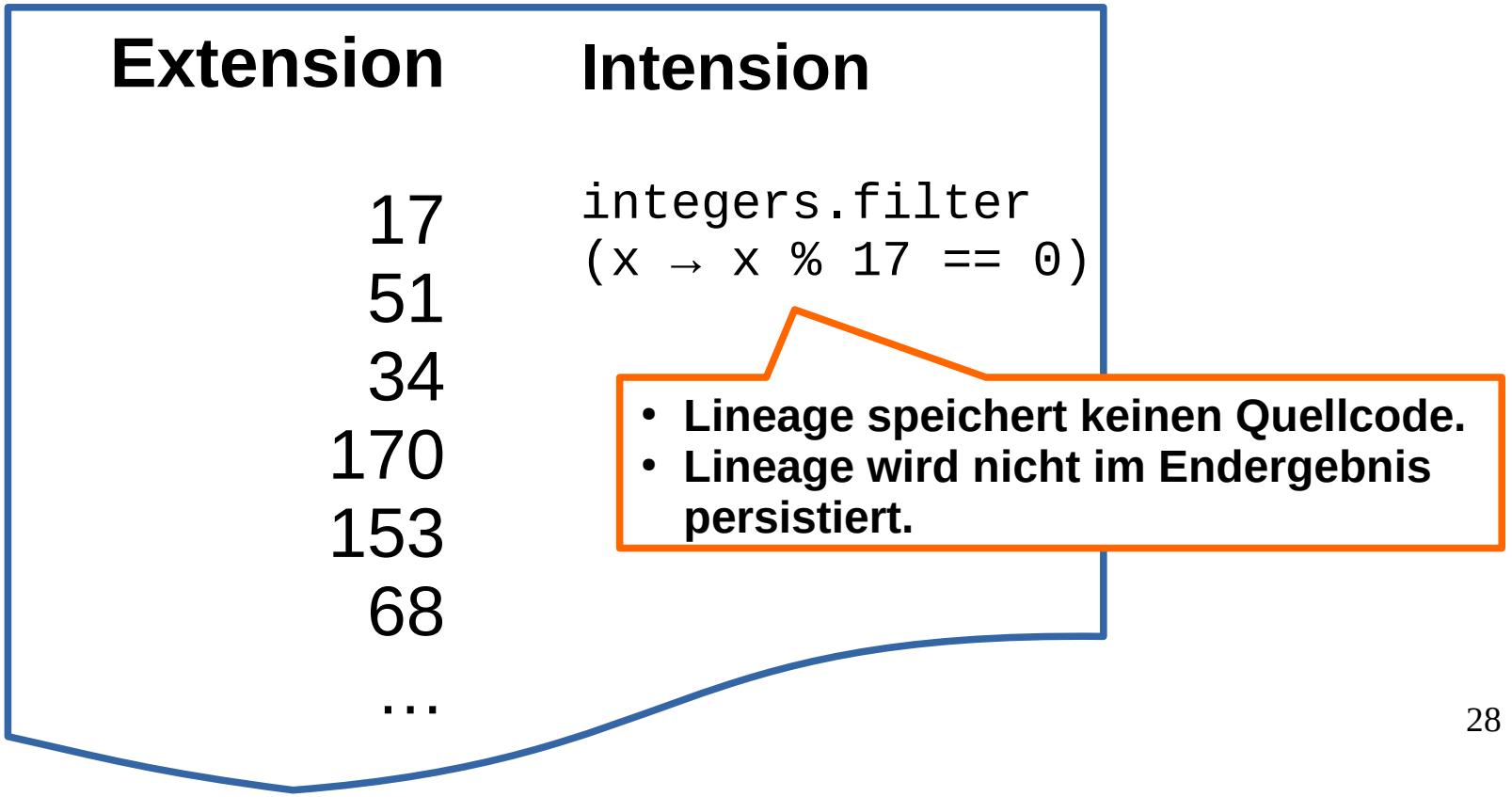
- Operationen liefern RDDs zurück, die Operationen anbieten, die RDDs liefern, die …

  → „Flüssige“ **Verkettung** von Operationen

```
long div2count = ctx.textFile("/tmp/data.txt")
        .map(Integer::valueOf)
        .filter(x -> x % 2 == 0)
        .count();
System.err.println(div2count);
```

Wie funktioniert Lineage? Closures

```
JavaRDD<Integer> integers = ...;
JavaRDD<Integer> filtered = integers.filter(x -> x % 17 == 0);
```



Wie funktioniert Lineage? Closures

```
JavaRDD<Integer> integers = ...;
Function<Integer, Boolean> func = x -> x % 17 == 0;
JavaRDD<Integer> filtered = integers.filter(func);
```

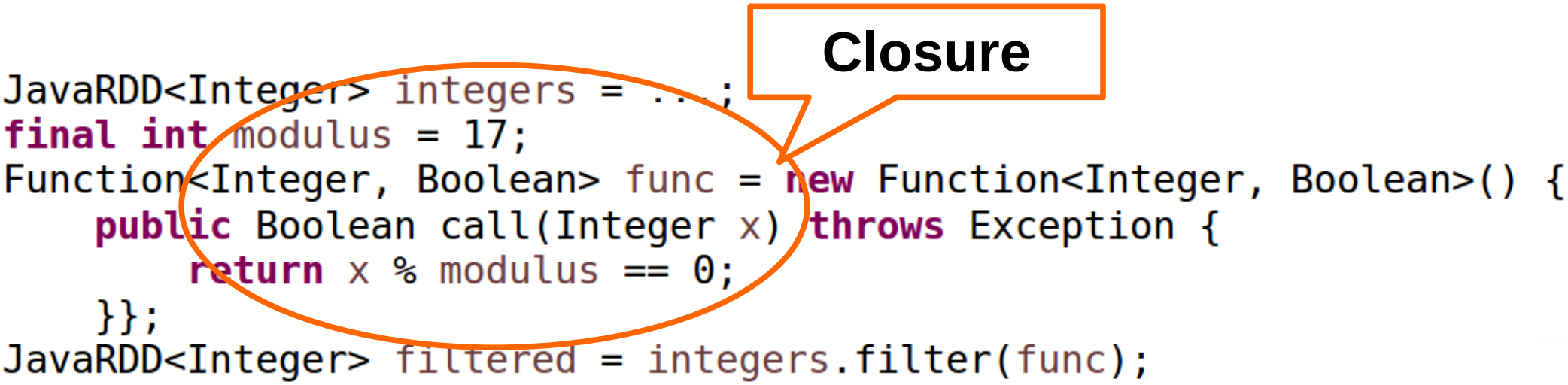
JavaRDD<T> filter(Function<T,Boolean> f)

Return a new RDD containing only the elements that satisfy a predicate.

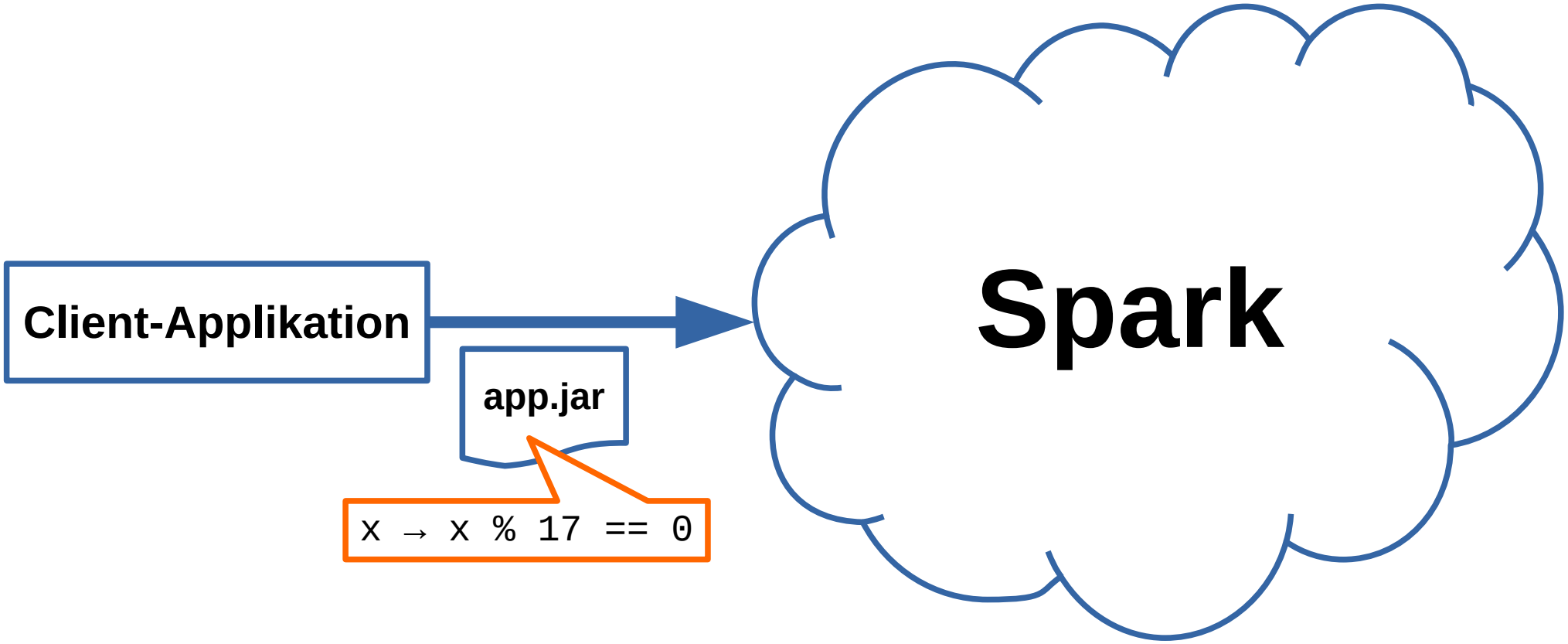
Wie funktioniert Lineage? Closures

```
JavaRDD<Integer> integers = ...;
Function<Integer, Boolean> func = new Function<Integer, Boolean>() {
    public Boolean call(Integer x) throws Exception {
        return x % 17 == 0;
    }};
JavaRDD<Integer> filtered = integers.filter(func);
```

Wie funktioniert Lineage? Closures

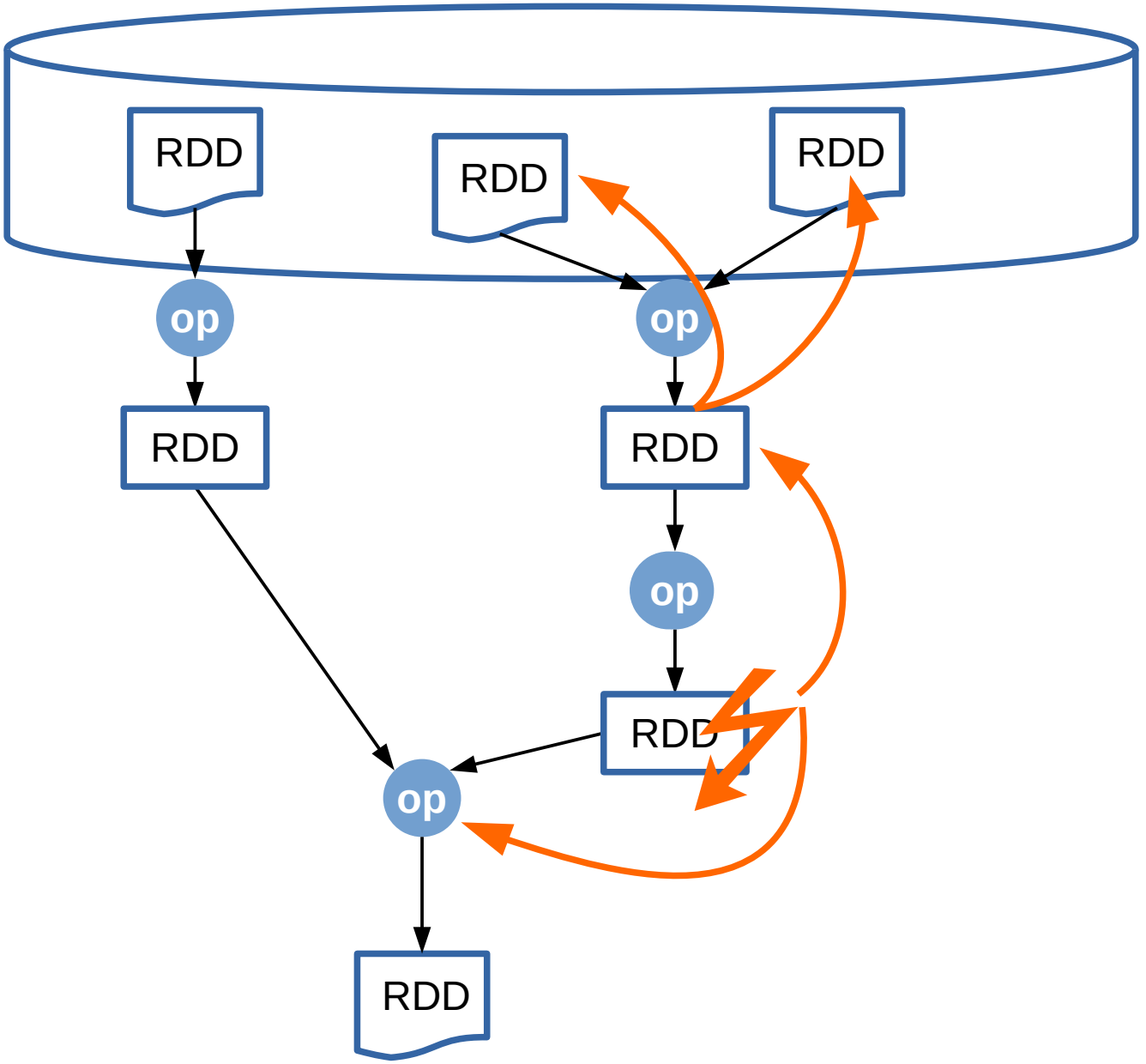


Wie funktioniert Lineage?



- Lineage nur innerhalb einer Client-Applikation
- Code (Closure) wird mit Job-Jar geliefert
- Code wird nicht mit RDD im HDFS persistiert

Lineage: Wozu das alles?



Verteilter Zustand in Spark

- Verteilter Zustand ist immer noch böse…

- **Broadcast-Variablen**
  - Verteilung von konstanten Daten

- **Akkumulatoren**
  - Verallgemeinerte Zähler

Beispiel: Broadcast-Variablen

```
Broadcast<List<Integer>> tableBroadcast =
        ctx.broadcast(ImmutableList.of(1, 3, 5));

JavaRDD<Integer> rdd = ...

rdd.map(i -> {
    List<Integer> table = tableBroadcast.getValue();
    ...
});
```

Beispiel: Akkumulatoren

```
LongAccumulator acc = sc.longAccumulator();

JavaRDD<Integer> rdd = ctx.parallelize(
        ImmutableList.of(1, 6, 2, 3, 4, 5, 3));
rdd.map(i -> {
    acc.add(i);
    return i * 2;
}).saveAsTextFile("...");

System.err.println(acc.value());
// --> 24
```

Zusammenfassung: Verteilter Zustand

- **Broadcast-Variablen**
  - nur lesender Zugriff
  - effiziente Verteilung

- **Akkumulatoren**
  - keine Verteilung während der Berechnung
  - kommutativ und assoziativ

➞ beide Konzepte unproblematisch!

Spark: Zwischenfazit

- Framework für **verteilte Berechnung**
- Kann **Hadoop-Infrastruktur** nutzen (HDFS/YARN)

- Abstraktion **RDD**
  - Extension und Intension (Daten und Herkunft)
  - Schmale und breite Transformationen
  - Komplexe Datenflüsse

- Konzepte aus **funktionaler Programmierung**
  - Referenzielle Transparenz
  - Lazy Evaluation
  - Unveränderliche Daten

Spark: Das Ökosystem

- **MLlib:** Machine Learning
- **GraphX:** Rechnen auf Graphen

- Spark **Streaming**
- Spark **SQL**
- **Data Frames**

Spark SQL

```
SELECT a, b, c, COUNT(*)
FROM t JOIN u
ON t.id = u.id
WHERE c > 117
GROUP BY b
HAVING COUNT(*) > 10
```

```
t.join(u)
 .filter(x → x.c > 117)
 .groupBy(x → x.b)
 .mapValues(
     Iterables::size)
 .filter
  (x → x._2 > 10)
```

→ **SQL-Operationen relativ leicht umsetzbar!**

Spark SQL: Beispiel

```
Dataset<Row> json = session.read().json("people.json");
System.out.println(json.schema());

json.createOrReplaceTempView("people");
List<Row> result = json.sqlContext()
        .sql("select * from people where age > 20")
        .toJavaRDD()
        .collect();
```

→ **StructType(StructField(age,LongType,true), StructField(name,StringType,true))**

→ **[[42,Alice]]**

```
{ "name": "Alice", "age": 42 }
{ "name": "Bob", "age": 14 }
{ "name": "Charlie", "age": 12 }
```

Spark SQL: Woher kommt das Schema?

- Quellen für Schema-Information
  - Metastore (Metadaten-Datenbank)
  - automatisch ableiten (z. B. bei JSON)
  - explizit bereitstellen

```
StructType schema = DataTypes.createStructType(
    new StructField[] {
        DataTypes.createStructField("name", DataTypes.StringType, true),
        DataTypes.createStructField("age", DataTypes.IntegerType, true)});
```

Data Frames

RDD

- API, Bibliothek
- für Entwickler

- Schema optional
- Intension/Extensio

Data Frame

- für Entwickler und Anwender
- Skripte, interaktiv

- Schema
- Intension/Extension
- Query-Operationen

SQL

- Query-Sprache
- für Endanwender

- Schema
- Intension

Data Frame d

`d[age]`
`d.age`

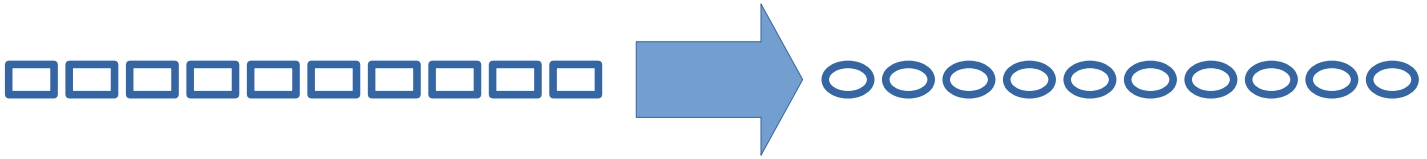
Name (String)	Age (Float)	Gender (m/w)	Height (Float)
Alice	33	w	1.72
Bob	26	m	1.83
Charlotte	49	w	1.56
David	46	m	1.79

`d[age > 40]`

```
d[30 <= age & age <= 40].agg({"age": "mean"}).show()

+----------+
| avg(age) |
+----------+
|       33 |
+----------+
```

Spark Streaming

- Ausblick → **Streaming-Kapitel**
- Verarbeitung von kontinuierlichen Strömen…



- … in **Micro-Batches:**