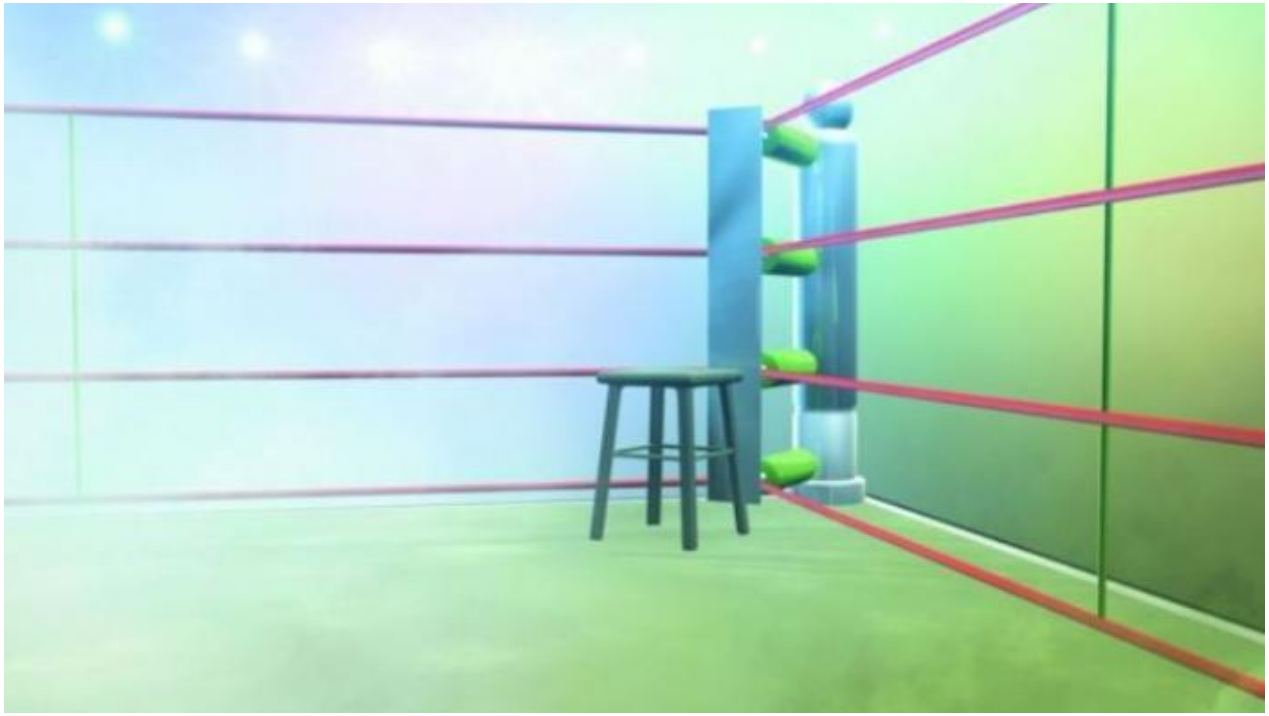
heise Developer

# Spark versus Flink – Rumble in the (Big Data) Jungle

**Know-how** 15.07.2016 09:18 Uhr

Michael Pisula, Konstantin Knauf



**Nachdem Apache Spark viele Sympathien erringen konnte und MapReduce langsam, aber sicher den Rang abläuft, kommt mit Apache Flink unerwartete Konkurrenz. Während Spark in erster Linie auf Batch-Verarbeitung setzt und durch Spark Streaming die Verarbeitung von Real-Time Daten mit Micro-Batching erlaubt, liegt bei Flink der Fokus auf Streaming, und alles andere baut darauf auf.**

Ende Januar 2016 feierte Apache Hadoop sein zehnjähriges Jubiläum. In dieser Dekade hat Hadoop maßgeblich zum Renommee von Big Data in der Softwarewelt beigetragen. Konnten Firmen vorher nur mit

Spezialhardware oder jahrelanger Forschung und Entwicklung große und schnell eintreffende Datenmengen verarbeiten, bot Hadoop auf Open-Source-Basis und Commodity-Hardware eine kostengünstige Möglichkeit, in die Big-Data-Verarbeitung einzusteigen. Die Grundbausteine von Hadoop, MapReduce und HDFS basieren auf Konzepten, die Google 2004 veröffentlichte und die Yahoo in Java schließlich unter dem Namen Hadoop implementierte.

Während HDFS immer noch eine wichtige Rolle bei Hadoop spielt, hat MapReduce seit der zweiten Hauptversion der Plattform an Bedeutung verloren. Inzwischen gibt es eine Vielzahl von Processing-Frameworks, die auf Hadoop beziehungsweise dem Resource Manager YARN laufen. Der Artikel vergleicht zwei miteinander, die beide unter dem Dach der Apache Software Foundation zu finden sind: **Spark[1]** und **Flink[2]**.

| | **Apache Spark** | **Apache Flink** |
|---|---|---|
| **Ursprung** | Berkeley University | TU Berlin |
| **Apache Incubator** | 2013 | 2014 |
| **Top-Level-Projekt** | 2/2014 | 1/2015 |
| **Firma hinter dem Projekt** | **Databricks[3]** | **Data Artisans[4]** |
| **APIs** | Scala, Java, Python, R | Java, Scala, Python |
| **Implementiert in** | Scala | Java |
| **Cluster** | Standalone, Mesos, EC2, YARN | Standalone, Mesos, EC2, YARN |

## Was ist Apache Spark?

Ursprünglich wurde Spark an der Berkeley University als Beispielapplikation für den dort entwickelten Ressourcen-Manager Mesos vorgestellt. Schnell stellte sich jedoch heraus, dass das Konzept mehr Aufmerksamkeit rechtfertigte. 2013 wurde Spark Teil des Incubator-Programms von Apache, in dem Projekte zeigen müssen, dass sie nach den Richtlinien der Open-Source-Organisation nachhaltig arbeiten. Im Februar 2014 erhielt die Technik dann die Weihen eines Top-Level-Projekts. Zu dem Zeitpunkt konnte das Framework bereits eine beeindruckende Anzahl Committer und Unterstützer aus der Wirtschaft vorweisen.

Im Vergleich zu MapReduce versprach Spark nicht nur eine um Faktor 100 schnellere Verarbeitung, sondern auch bessere Testbarkeit und höheren Entwicklungskomfort. Zudem war Spark von Anfang an modular konzipiert, sodass sich neben der Batch-Verarbeitung Streaming, Machine Learning und Graph Processing

innerhalb des Frameworks umsetzen lassen. Besondere Bedeutung hat dabei die Stream-Verarbeitung, da man dafür bisher auf andere Tools wie Apache Storm zurückgreifen musste. Das ist gerade bei der Verwendung sogenannter Lambda-Architekturen wichtig, bei denen sich Daten sowohl über schnelle Streaming-Analysen als auch langsamere Batch-Prozesse verarbeiten lassen. Spark ermöglichte es, beides in einem Framework zu realisieren und die Datenströme zu kombinieren. Des Weiteren erlaubt es Spark interaktive Analysen durchzuführen, so kann man schnell die Daten kennenlernen und Prototypen von Analysen erstellen.

## Was ist Apache Flink?

Auch Flink entstand an einer Universität, allerdings an einer deutschen, der TU Berlin. Es ging dort aus dem Forschungsprojekt Stratosphere hervor. Flink bietet ebenfalls mehrere Möglichkeiten der Datenverarbeitung: Batch, Streaming, Machine Learning und Graph Processing. Ein Unterschied ist aber die Ausrichtung. War der Fokus von Spark von Anfang an eine schnellere Batch-Verarbeitung, spezialisiert sich Flink auf die Verarbeitung von kontinuierlichen Datenströmen, also Stream Processing. Flink ist im April 2014 ins Incubator-Programm der Apache Software Foundation aufgenommen worden und wurde im Januar 2015 zum Top-Level-Projekt.

Flink hat derzeit noch weniger Committer als Spark zu verzeichnen, bekommt in letzter Zeit aber viel Schub. Zum einen ist dafür die vorteilhafte Performance im Vergleich zu Spark verantwortlich, zum anderen, dass viele Konzepte, die Spark erst noch einführt, in Flink von Anfang an vorhanden waren (etwa Back Pressure oder ein selbstverwalteter Off-Heap-Speicherbereich). Auch der Fokus auf Streaming ist für viele wichtig: Im Vergleich zu Storm bietet Flink bessere Performance, eine deutlich schlankere API und modernere Konzepte.

## Batch Processing

## Umgang beim Batch Processing

Im weiteren Verlauf werden die wichtigsten Konzepte beider Frameworks anhand eines einfachen Beispiels vorgestellt. Um zu verdeutlichen, wie Programme mit Spark beziehungsweise Flink aussehen, wird der komplette Code gezeigt. Zur besseren Vergleichbarkeit und wegen der größeren Verbreitung nutzen die Autoren die Java APIs beider Frameworks. Wer keine Berührungsängste zu Scala hat, sollte aber auf jeden Fall einen Blick auf die Scala APIs riskieren. Konzepte wie Case Classes und Pattern Matching machen den Scala-Code um einiges kürzer und eleganter als den in Java.

Das Beispiel zur Verarbeitung von Batch-Daten besteht daraus, eine CSV-Datei einzulesen, die eine Liste verkaufter Artikel enthält. In der Datei gibt es die Artikel-ID, die Anzahl verkaufter Artikel und das Verkaufsdatum. Aus den Daten soll abgeleitet werden, welcher Artikel pro Tag am häufigsten verkauft wurde. Das Beispiel ist relativ einfach gehalten, erlaubt aber einen Blick auf viele wichtige Operatoren.

In der Batch API von Spark gibt es ein Kernkonzept, das Resilient Distributed Dataset (RDD). Dabei handelt es sich um eine Datenstruktur, die sich zum einen auf einen Cluster verteilen lässt (distributed) und zum anderen Ausfallsicherheit bietet (resilient). Die Ausfallsicherheit ist so umgesetzt, dass man bereits berechnete Daten wieder aus den Quelldaten nachberechnen kann. Fällt ein Cluster-Knoten aus, übernehmen die anderen die Neuberechnung der verlorenen Daten. Mit Spark lässt sich auch ein Zwischenstand der Daten persistieren, womit dieser Datensatz für Nachberechnungen der neue Ausgangszustand wird.

In den letzten Versionen hat Spark neben dieser Abstraktion noch zwei weitere Batch APIs eingeführt, die Dataframes und die Datasets. Insbesondere Erstere bringen eine höhere Geschwindigkeit, jedoch zu Lasten der Typsicherheit. Die Dataset API ist erstmals in Spark 1.6 zu finden und soll mehr Typsicherheit als die Dataframes bringen, bei besserer Performance als bei den RDDs. Derzeit empfiehlt es sich allerdings, noch die zwei APIs aus Scala zu nutzen; bei beiden gibt es noch einige Einschränkungen in der Java API. Der Artikel verwendet die RDDs, da sie die historische Basis von Spark sind, aber auch weil andere Teile von Spark, etwa Streaming, darauf aufbauen. Es ist allerdings absehbar, dass die DataFrames und Datasets APIs die RDD API auch dort verdrängen werden.

**Codebeispiel: Einlesen und Verarbeiten einer Datei**

```
SparkConf conf =
      new SparkConf().setAppName("Spark Demo").setMaster("local[*]");
JavaSparkContext sparkContext = new JavaSparkContext(conf);

JavaRDD<String> salesFile = sparkContext.textFile(salesPath);
JavaPairRDD<Sale,Integer> salesWithAmount = salesFile
      .map(line -> line.split(",")) // Splitten des CSV in einzelne Felder
      .filter(array -> array.length == 3) // Rausfiltern ungültiger Zeilen
      .mapToPair(parts -> { // Umwandeln der Werte in einen POJO
          int id = Integer.parseInt(parts[0]), amount =
```

```
                    Integer.parseInt(parts[1]);
            String date = parts[2];
            return new Tuple2<Sale,Integer>(new Sale(id, date), amount);
        });

// Addieren der einzelnen verkauften Mengen
JavaPairRDD<Sale, Integer> saleWithSum =
                    salesWithAmount.reduceByKey((x, y) -> x + y);

saleWithSum.rdd().toJavaRDD() // Umwandeln von JavaPairRDD nach JavaRDD
        .sortBy(Tuple2::_2, false,1) // Sortieren nach der Menge
        .foreach(tuple -> System.out.println(tuple._1() + ":
                    " + tuple._2())); // Ausgeben der Ergebnisse
```

Zu Beginn ist bei Spark ein *SparkContext* zu erstellen. Er lässt sich mit der *SparkConf* konfigurieren. So kann man den Namen der Applikation setzen, Speichergrößen festlegen, und vor allem den Master bestimmen. Er legt fest, wo das Programm ausgeführt wird – auf einem Cluster oder lokal in der JVM. Im Cluster-Fall gibt man hier die Adresse des Masters ein, ansonsten wie im Beispiel "local". Dahinter kann man noch die Parallelität festlegen. "*" bedeutet, dass Spark nach Belieben parallelisieren kann.

Spark erlaubt das Konsumieren von Daten aus verschiedenen Quellen: HDFS (Hadoop Distributed File System), S3, JDBC (Java Database Connectivity), aber auch aus normalen Textdateien wie im Beispiel. Entsprechend gibt es verschiedene Methoden, die für eine Datenquelle einen RDD zurückgeben, hier kommt *textFile* zum Einsatz. Wer einen RDD hat, kann auf ihm zwei Typen von Operationen ausführen: Transformationen (*map*, *reduce*, *filter* ...) und Aktionen (*save*, *print* ...). Erstere sind "lazy" und werden nur ausgeführt, wenn am Ende der Aufrufkette eine Aktion steht. So kann man die Aktion auskommentieren, und die ganze Operatorenkette wird nicht mehr durchgeführt.

Spark bietet für die RDD API keine Möglichkeit, CSV- oder JSON-Daten einfach einzulesen. Hier müssen Entwickler entweder, wie im obigen Beispiel, selbst die Daten extrahieren oder auf andere Libraries dafür zurückgreifen. Die Dataframes API bietet hingegen etwas mehr Komfort; unter anderem gibt es ein Plug-in zum direkten Einlesen von CSV-Dateien.

Da die Menge pro Artikel und Tag berechnet werden soll, fassen die Autoren diese zwei Felder in einem POJO (Plain Old Java Object) zusammen. Da sie immer zusammen behandelt werden, macht das den Code einfacher und aufgeräumter. Die verkaufte Menge kann nicht Teil des POJOs sein, da über diesen Wert summiert werden soll. Um alle Werte für einen Schlüssel (hier das *Sale*-Objekt) zu bekommen, benutzen Entwickler den *reduceByKey*-Operator. Er ist nur auf einem RDD über Tupel definiert. In der Java API gibt es dafür den Typ *JavaPairRDD*, den man über die *mapToPair*-Methode bekommt. Sie erwartet als Rückgabe aus dem Lambda ein *Tuple2*-Objekt. Es ist eine in Scala nativ vorliegende Klasse, die in der Spark API fest verwurzelt ist. Die *mapToPair*-Methode gibt es nur in der Java API, unter Scala stehen die *byKey*-Methoden automatisch für RDDs über ein Tuple zur Verfügung.

Auf dem *JavaPairRDD* lässt sich nun *reduceByKey* aufrufen. In einem Lambda wird übergeben, wie mit den Werten verfahren werden soll. In diesem Fall wird einfach aufsummiert. Damit gruppiert Spark nun den RDD nach gleichen Sale-Objekten und summiert die dazugehörigen Mengen. In der Verarbeitung im Cluster ist das eine potenziell kostspielige Operation. Bisher ließen sich alle Operationen (*map* und *filter*) für jeden Datensatz unabhängig, und damit parallel, voneinander durchführen. Für den *reduce*-Schritt werden nun aber Datensätze für die gleichen Schlüssel auf demselben Knoten benötigt. Im Cluster ist hierzu neu zu partitionieren. Das heißt, die Daten werden zwischen den Nodes hin und her kopiert. Bei großen Datenmengen und vielen Cluster-Knoten sollte man nicht ohne Not solche Operationen einsetzen.

Die Verarbeitung findet ihren Abschluss, indem die Daten erst sortiert und dann auf die Konsole ausgegeben werden. Da man auf dem JavaPairRDD nur nach dem Key sortieren kann, geschieht die Umwandlung über den Umweg des Scala RDD in den Java RDD.

### Bei Flink ähnlich

Die DataSet API von Flink bietet ähnliche Funktionen wie Spark, doch beim Einlesen von Dateien verfügt die Technik über bequemere Funktionen. So lassen sich CSV-Dateien einfach einlesen und auf POJOs oder Tupel abbilden. Auch für häufig genutzte *reduce*-Funktionen bietet Flink eigene Methoden an, sodass der Code deutlich kürzer und verständlicher ausfällt.

```
final ExecutionEnvironment env =
     ExecutionEnvironment.getExecutionEnvironment();
```

```
env.readCsvFile(salesPath)
      .ignoreInvalidLines()
      .types(Integer.class, Integer.class, String.class)
      .groupBy(0,2) // Gruppieren nach ArtikelID und Datum
      .sum(1)       // Verkaufszahlen in diesen Gruppen aufsummieren
      .groupBy(2)   // Gruppieren nach Tag
      .maxBy(1)     // Pro Gruppe (also Tag) Artikel mit den meisten Sales
      .print();
```

Bei Flink gibt es ebenfalls zwei Typen von Operatoren, sodass die Verarbeitung nur geschieht, wenn die Ergebnisse auch benötigt werden. Im Gegensatz zu Spark muss man bei Flink nicht im Code einstellen, wo und wie die Applikation laufen wird. Flink erkennt selbst, ob es lokal oder im Cluster läuft.

Auf *Environment* lassen sich auch Daten aus verschiedenen Quellen einlesen. Wie erwähnt gibt es hier aber eine Möglichkeit, CSV-Dateien zu verarbeiten. Da sie sich oft nur lose an den Standard halten, gibt es viele Mittel, bestimmte Parameter zu setzen, etwa den Separator oder eben, ob ungültige Zeilen gefiltert werden sollen. Zum Schluss lassen sich die Daten direkt in einem POJO umwandeln. In dem Fall werden die Felder einfach auf ein Tupel abgebildet. Hierzu ist nur anzugeben, welche Typen für die Felder zum Einsatz kommen. Als Ergebnis erhält man ein *DataSet*-Objekt, auf dem ähnliche Operationen möglich sind wie vorher bei Spark auf dem RDD.

Auch gibt man an, dass die ID und das Datum als Schlüssel genutzt werden sollen. In Flink erledigt man das mit *groupBy* und gibt beide relevanten Felder an. Danach kommt die *sum*-Funktion zum Einsatz, die auf einem *GroupedDataSet* definiert ist. Diese und einige weitere Funktionen werden als spezielle *reduce*-Funktionen direkt in der API angeboten, damit sich Benutzer etwas Code und Komplexität sparen können. Framework-seitige Optimierungen geraten so ebenfalls einfacher. Zum Schluss werden die Daten noch sortiert und ausgegeben.

Auch wenn die Lösung in beiden Frameworks ähnlich aufgebaut ist, bietet Flink subjektiv gesehen die schönere, ausdrucksstärkere API und mehr Komfort. Mit den neuen APIs wird Spark hier aber aufholen.

## Streaming

## Vorgehen beim Streaming

In vielen Geschäftsbereichen kann Geschwindigkeit in der Datenanalyse und -verarbeitung einen Wettbewerbsvorteil bedeuten. Insbesondere für viele Internet-Unternehmen ist schnelle Datenverarbeitung Kerngeschäft. Doch nicht nur, wenn es um Geschwindigkeit geht, kann man von Streaming profitieren, auch Anwendungsfälle, die derzeit mit Batch-Verarbeitung gelöst sind, lassen sich teilweise als Streaming-Anwendung einfacher und performanter darstellen.

Dennoch ist es kein Wunder, dass wichtige Tools im Real-Time-Bereich wie Storm oder Samza von Twitter beziehungsweise LinkedIn stammen. Hier gibt es aber noch Bedarf für ein Tool, das alle Ansprüche erfüllt. Lange war Storm Marktführer, es scheint aber inzwischen an Fahrt zu verlieren. Twitter hat sich inzwischen von der Technik abgewandt und mit **Heron[5]** ein neues Tool geschaffen.

Deshalb ist es interessant, einen Blick auf Flink zu werfen, da es eine echte Real-Time-Plattform mit vielen weiteren Features bietet und sich durchaus zum Branchenprimus aufschwingen kann. Dabei wird Flink zugute kommen, dass es – wie Spark – die komplette Bandbreite von Batch über Graph Processing und Stream Processing bis hin zu Machine Learning anbietet.

Bei der Batch-Verarbeitung war der Code bei den Flink- und Spark-Programmen konzeptionell noch ähnlich; das ändert sich nun bei der Verarbeitung von Streams. Grund dafür ist, dass Spark kein reines Streaming beherrscht, sondern die Daten in kleinen Zeitabständen zu Batches zusammenfasst – das Konzept nennt sich Micro-Batching. Darunterliegend sind bei Spark dann immer noch die RDDs, mit denen man auch beim Streaming noch Kontakt hat. Bei Flink gibt es eine komplett separate API für Streaming, die DataStream API. Der funktionale Ansatz bei der Datenverarbeitung (*map*, *reduce* etc.) gilt aber weiterhin für beide Frameworks.

Bei der Verarbeitung von Streams werden die Daten in der Regel aus einer Queue eingelesen. Diese erfüllt im Wesentlichen zwei Funktionen. Erstens entkoppelt sie Datengenerierung und -verarbeitung und kann Daten zwischenspeichern, falls die Verarbeitung nicht mit der Generierung mithalten kann. Zweitens lassen sich Daten leicht erneut in die Verarbeitung einspeisen, falls es zu Fehlern oder Ausfällen bei der Verarbeitung gekommen ist. Im Big-Data-Umfeld hat sich **Apache Kafka[6]** als verteilte Queue zum Standard-Tool entwickelt, weshalb die folgenden Beispiele Kafka nutzen.

## Streaming bei Flink

Für Real-Time Streaming hat Flink eine eigene API namens DataStreams. Sie stellt ganz ähnliche Operationen

wie die DataSet API zur Verfügung, zusätzlich aber noch einige weitere, die speziell auf den Streaming-Fall ausgelegt sind. Besondere Bedeutung kommt bei der Stream-Verarbeitung der Definition von Windows auf dem Stream zu. Denn in einem Stream kann man normalerweise immer nur ein einzelnes Event betrachten. Abgesehen von einfachen Filter- und Umformatierungsoperationen lässt sich damit noch nicht viel anfangen.

Interessanter wird es bei der gleichzeitigen Betrachtung von Events über einen gewissen Zeitraum. So könnte es erforderlich sein, gewisse Analysen auf allen Events durchzuführen, die in den letzten fünf Minuten angekommen sind. Ungleich spannender ist die Möglichkeit, solche Fenster auch auf den Zeitstempeln der Events zu definieren. Somit kann man etwa alle Events, die in einer bestimmten Stunde erzeugt wurden, gesammelt verarbeiten, selbst wenn sie über den Zeitraum von vier Stunden angeliefert wurden (bspw. wegen Ausfällen in einer Zwischenkomponente oder wegen eines Backlogs). Gerade bei Auswertungen über so lange Zeiträume ist es wichtig, einen internen Status persistent speichern zu können.

Zur Veranschaulichung der Stream API kommt ein ähnliches Beispiel zum Einsatz wie im Batch-Fall: Aus einem Strom von Sale-Events ist zu bestimmen werden, welcher Artikel für welchen Tag wie oft verkauft wurde.

```
StreamExecutionEnvironment env =
         StreamExecutionEnvironment.getExecutionEnvironment();

DataStreamSource<String> kafkaSource = env.addSource(new
         FlinkKafkaConsumer08<>("topic",
                 new SimpleStringSchema(), kafkaProperties));

kafkaSource.map(line -> line.split(","))
           .map(new MapFunction<String[], Tuple3<Integer, String,
                 Integer>>() {
                @Override
                public Tuple3<Integer, String, Integer> map(String[] parts)
                        throws Exception {
                    int id = Integer.parseInt(parts[0]), amount =
                           Integer.parseInt(parts[1]);
                    String date = parts[2];
                    return new Tuple3<Integer, String, Integer>(id, date,
```

```
                        amount);
                }
            })
            .keyBy(0, 1)
            .sum(2)
            .print();

env.execute(); //Start processing
```

Der erste Unterschied zum Batch-Code ist die Verwendung der *StreamExecutionEnvironment*, die Zugang zur Streaming API gewährt. Auf ihr fügt man mit *addSource* die Datenquellen hinzu. Es gibt, insbesondere zum Testen und Ausprobieren, einige fertige Quellen, die man über Methoden auf *StreamExecutionEvironment* bekommt, etwa *readTextFile()*, das Daten aus einer Datei liest und als Stream verarbeitet. Für das Beispiel wird Kafka als Quelle verwendet. Dazu liefert Flink einen Kafka-Consumer mit, derzeit sowohl für Kafka 0.8 als auch Kafka 0.9. An Unterstützung für Kafka 0.10 arbeitet das Team aktuell.

Um die Binärdaten aus Kafka als String zu deserialisieren, nutzen die Autoren das *SimpleStringSchema*, das Flink ebenfalls bereitstellt. Den Kafka Topic sowie einige Kafka-Einstellungen (etwa die Broker bzw-Zookeeper-Adresse) muss man hier ebenfalls übergeben. Daraus ergibt sich nun eine *DataStreamSource*, die von der Hauptklasse der Streaming API (*DataStream*) erbt und somit den Zugriff auf die Streaming API bereitstellt. Im Gegensatz zur Batch API liefert Flink keine eingebaute Methode, mit CSV umzugehen, deshalb ist hier manuell zu splitten.

Der Rest der Verarbeitung ist ähnlich wie zuvor im Batch-Code. Diesmal wird die Funktion *keyBy* verwendet, nicht *groupBy*. Sie liefert einen *KeyedDataStream* der nach den angegebenen Schlüsseln partitioniert ist. Den praktischen Aggregation-Operator *sum* gibt es auch hier. Er bewirkt, dass in dem gerade betrachteten Event das angegebene Feld durch den bisher aggregierten Wert ersetzt wird.

Je mehr Nachrichten für einen bestimmten Schlüssel eingehen, desto größer wird demnach der Wert. Damit ist die Umsetzung abgeschlossen. Wenn Events in die Kafka Queue geschrieben werden, wird ausgegeben, wie viele Artikel bisher pro Tag und Artikel verkauft wurden.

### Streaming bei Spark

Bei Spark Streaming gibt es kein echtes Real-Time. Stattdessen fasst es die empfangenen Events alle 0,5 bis 2 Sekunden in sogenannte Micro-Batches zusammen. Das hat den großen Vorteil, dass Spark mit Batching gut umgehen und man so sehr viele Operationen wiederverwenden kann. Zudem lassen sich Streaming- und Batch-Daten einfach kombinieren; es sind im Grunde ja beides RDDs. Es hat aber den Nachteil, dass es eine gewisse Latenz mit sich bringt. Dem Vorgehen fallen auch einige Windowing-Arten zum Opfer. Während Windows auf Processing Time einfach umzusetzen sind (zumindest solange das Fenster ein Vielfaches der Micro-Batch-Länge ist), sind Event-Time Windows oder Windows mit fester Anzahl an Events mit Spark derzeit nicht trivial möglich.

Das gleiche Beispiel wie oben sieht in Spark wie folgt aus:

```
SparkConf conf = new SparkConf().setAppName("Spark
      Demo").setMaster("local[*]");
JavaStreamingContext streamContext = new JavaStreamingContext(conf,
      Durations.seconds(1));

JavaPairDStream<Sale,Integer> aggregatedSales =
      KafkaUtils.createDirectStream(streamContext, String.class,
               String.class, StringDecoder.class, StringDecoder.class,
                     kafkaParams, topics)
      .map(tuple -> tuple._2);
      .map(line -> line.split(","))
      .filter(array -> array.length == 3) // Rausfiltern ungültiger Zeilen
      .mapToPair(parts -> { // Umwandeln der Werte in einen POJO
          int id = Integer.parseInt(parts[0]), amount =
                   Integer.parseInt(parts[1]);
          String date = parts[2];
          return new Tuple2<Sale,Integer>(new Sale(id, date), amount);
      })
      .reduceByKey((x, y) -> x + y);

aggregatedSales.foreachRDD(rdd -> {
```

```
    rdd.map(tuple->tuple).sortBy(tuple -> tuple._2, false, 1)
            .take(1)
            .forEach(System.out::println);
});

streamContext.start();
streamContext.awaitTermination();
```

Der Ausgangspunkt ist ein *StreamingContext*. Für ihn ist eine Duration zu übergeben. Durch das Micro-Batching gibt sie an, wie lang diese Batches sein sollen. Mit der Einstellung wird Spark jede Sekunde ein neues Paket mit Daten erzeugen.

Auch bei Spark ist die Datenquelle Kafka. Die *KafkaUtils*-Klasse von Spark, mit der sich Kafka als Quelle und Senke verwenden lässt, bietet verschiedene Methoden an. Die hier verwendete Methode *createDirectStream* ist die momentan empfohlene. Derzeit unterstützt Spark nur Kafka 0.8, an der Kafka-0.10-Unterstützung wird gearbeitet.

Die Spark API gibt in einem Kafka Stream sowohl den Schlüssel als auch den Wert. Wie bei Flink ist hier anzugeben, wie man aus den Kafka-Nachrichten die gewünschten Rückgabetypen extrahieren kann. Die verwendete Klasse *StringDecoder* stellt Kafka bereit. Wie bei Flink sind der Topic und einige Kafka-Einstellungen anzugeben. Als Ergebnis bekommt man einen *JavaDStream* und damit Zugriff auf die Operatoren, die sich auf dem Stream aufrufen lassen.

Zuerst entfernen Entwickler den Key, da sie nur mit dem Value weiterarbeiten sollen. Dann wird analog zum Batch-Beispiel das CSV geparst und ein POJO erstellt. Danach ruft man erneut *reduceByKey* auf, gefolgt von der Ausgabe der Daten. Der Code ist fast der gleiche wie im Batch-Beispiel. Vor dem Sortieren und Ausgeben muss man jedoch *foreachRDD* aufrufen. Damit erhalten Entwickler die RDDs, also Batches, aus denen der DStream aufgebaut ist. Darauf hin lässt sich einfach auf die Operatoren zurückgreifen, die nur auf RDDs verfügbar sind, etwa das Sortieren. Auf einem Stream ergibt der Operator keinen großen Sinn, auf den einzelnen Batches kann aber sinnvoll sortiert werden. Die Tatsache, dass Spark Micro-Batching nutzt, tritt so in den APIs recht häufig zu Tage.

Damit bekommt man nun für jeden Ein-Sekunden-Batch die Summe der darin verkauften Artikel pro Tag. Das

entspricht nicht ganz der Aufgabenstellung. Um die Summe über einen längeren Zeitraum zu persistieren, muss man in Spark deutlich mehr Aufwand betreiben. Das aktuelle Spark 1.6 führte eine neue Methode ein, mit der man sich einen Zustand merken kann. Der vollständige Code würde hier aber den Rahmen sprengen und lässt sich im **GitHub-Repository zum Artikel[7]** nachschauen.

Mit der kommenden Version 2.0 von Spark wird sich einiges ändern. Der neue Ansatz für Streaming soll deutlich schlanker sein und auf den neuen Dataframes/Dataset APIs aufsetzen. Hier wird die Zeit zeigen, ob Spark mit der neuen API im Bereich Streaming zu Flink aufschließen kann.

## Fazit

Beide Tools sind dafür geeignet, komplexe Big-Data-Applikationen mit verschiedenen Arten der Verarbeitung umzusetzen. Für Spark spricht, dass es stark auf dem Markt vertreten, in vielen Hadoop-Distributionen integriert ist und mehrere große Firmen hinter sich vereinen kann. Innovationen kommen mit jedem neuen Release dazu. So ist nicht zu befürchten, dass Spark bald die Puste ausgeht. Es gibt aber durchaus einige konzeptionelle Altlasten, die das Framework mit sich herumträgt.

Flink scheint nicht nur dort auf einem solideren Fundament aufgebaut zu sein, auch die API, das Monitoring, in vielen Bereichen wirkt Flink einfach frischer. Durch seine europäischen Wurzeln kämpft es aber durchaus um Akzeptanz bei den großen, in Nordamerika ansässigen Firmen. Während etwa auf der Strata, der wohl bedeutendsten Big-Data-Konferenz, ein kompletter Track im Zeichen von Spark steht, ist Flink mit nur einem Vortrag vertreten. Dass das bei europäischen Konferenzen durchaus anders ist, macht aber Hoffnung. Auch dass bei der letztjährigen Flink-Forward-Konferenz in Berlin schon einige Produktionsszenarien zu der noch jungen Technik vorgestellt wurden, ist ein gutes Zeichen.

Die Entscheidung für oder wider eines der beiden Tools ist nicht einfach. Wenn man Real-Time Streaming braucht, aber auch Batch Processing, ist Flink eine ausgezeichnete Wahl. Ansonsten ist die Frage relevant, ob man auf die etablierte Technik setzt oder ein Early Adopter sein möchte. Die eingesetzte Programmiersprache kann durchaus auch eine Rolle spielen. Spark ist vorrangig in Scala geschrieben, und die Java API wird von der Scala API abgeleitet. Dass es nicht immer ganz einfach ist, von der mächtigeren Sprache abzuleiten, wird schnell deutlich. Einige Teile der API sind unschön.

Flink ist zu großen Teilen in Java geschrieben und hat eine deutlich stimmigere Java API. Bei Data-Analysten

erfreut sich Python auch großer Beliebtheit – hier hat Spark eindeutig die Nase vorn. Zwar verfügt auch Flink über eine Python API, die von Spark ist aber deutlich ausgereifter.

Performance ist für viele Big-Data-Anwendungen ein zentrales Thema und soll hier nicht ignoriert werden. Bei der Batch-Verarbeitung gibt es einige klassische Benchmarks wie TeraSort, zwischen Flink und Spark [**1[8]**] [**2[9]**]. In den meisten ist Flink Spark bei der Laufzeit überlegen. Das wird hauptsächlich auf das stärkere Pipelining zwischen den Schritten zurückgeführt. Hier wird Spark mit den neuen APIs wohl aufholen können, aktuell fehlen hier aber noch belastbare Zahlen.

Bei der Stream-Verarbeitung gilt es, zwei Performance-Aspekte zu betrachten: Durchsatz (Records/s) und Latenz. In Sachen Latenz ist Flink Spark per Design überlegen, da Spark durch das Micro-Batching eine Latenz in Höhe des Batch-Intervalls hinzufügt. Beim Thema Durchsatz gibt es leider noch keine verlässlichen Benchmarks. **In einem von Yahoo[10]** kommen beide auf die höchste getestete Performance von 170.000 Records/s. Eine verbesserte Variante der getesteten Flink-Applikation **erreichte bei Twitter[11]** sogar 15.000.000 Records/s. Unklar ist hier, wie viel man durch Tuning bei Spark noch hätte verbessern können. Da Spark kein reines Streaming-Framework ist, gibt es generell mehr Streaming-Benchmarks, die Flink mit Apache Storm vergleichen. In ihnen schlägt Flink Storm regelmäßig um Größenordnungen; insbesondere wenn bei Storm Verarbeitungsgarantien (das sogenannte Acking) aktiviert sind. Hierfür hat Flink ein leichtgewichtigeres Konzept gewählt. Mit der kürzlich erschienenen Version 1.0 von Apache Storm hat sich in diesen Punkten viel getan. Es lohnt sich also, die Augen nach neuen Benchmarks offen zu halten.

Ein anderes Kriterium, das insbesondere in großen Firmen oft eine Rolle spielen kann, ist das Thema Support. Spark wird inzwischen in allen großen Hadoop-Distributionen mitgeliefert und unterstützt. Auch Databricks, die Firma hinter Spark, bietet Support an. Bei Flink wird Data Artisans wohl über kurz oder lang auch Produktionssupport anbieten, noch ist man allerdings auf die (allerdings sehr aktive und hilfsbereite) Mailingliste angewiesen.

Ein großer Vorteil, wenn es um die Evaluierung der Tools geht, ist, dass beide auf der gleichen Plattform laufen. Wer einen Hadoop-Cluster hat, kann Prototypen in beiden Tools entwickeln und so entscheiden, welches der beiden sich für den Einzelfall besser eignet. (**ane[12]**)

*Michael Pisula*
*hat Informatik an der Universität Passau studiert. Sein besonderes Interesse gilt den verteilten Systemen,*

*insbesondere der immer wichtiger werdenden Big-Data-Welt. Als Senior Consultant bei TNG hilft er Kunden, wenn es um Big Data, Akka, Continuous Integration und allgemein um nichttriviale Probleme geht.*

*Konstantin Knauf*
*hat an der TU Darmstadt Mathematik und Informatik mit Schwerpunkt Machine Learning studiert. Als Software Consultant bei TNG Technology Consulting unterstützt er Kunden vorrangig in den Bereichen Big Data und Automatisierung.*

---

**URL dieses Artikels:**

`http://www.heise.de/-3264705`

**Links in diesem Artikel:**

**[1]** `http://spark.apache.org/`

**[2]** `https://flink.apache.org/`

**[3]** `https://databricks.com`

**[4]** `http://data-artisans.com/`

**[5]** `https://blog.twitter.com/2015/flying-faster-with-twitter-heron`

**[6]** `http://kafka.apache.org/`

**[7]** `https://github.com/michaelpisula/spark-flink`

**[8]** `http://shelan.org/blog/2016/01/31/reproducible-experiment-to-compare-apache-spark-and-apache-flink-batch-processing`

**[9]** `http://eastcirclek.blogspot.de/2015/06/terasort-for-spark-and-flink-with-range.html`

**[10]** `https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at`

**[11]** `http://data-artisans.com/extending-the-yahoo-streaming-benchmark/`

**[12]** `mailto:ane@heise.de`