



# Masterstudiengang Wirtschaftsinformatik

# Big Data Engineering

FH Münster  
Master Wirtschaftsinformatik  
Wintersemester 2013  
Dozent: Lars George



# Einheit 7

- Rückblick auf Einheit 6
- Hybride Architekturen
  - Batch und Real-time gemischt
- „Best Practices“ für Clusteraufbau
- BI Integration
- **Hauptziel:** Abschließende Kenntnisse für den Einsatz von Big Data Systemen kennenlernen.
- **Übung 7:**
  - Arbeit an Projekt



# Einheit 7

- **Rückblick auf Einheit 6**
- Hybride Architekturen
  - Batch und Real-time gemischt
- „Best Practices“ für Clusteraufbau
- BI Integration



# Rückblick auf Einheit 6

- Fragen?
  - Oozie ausprobiert?



# Einheit 7

- Rückblick auf Einheit 6
- **Hybride Architekturen**
  - Batch und Real-time gemischt
- „Best Practices“ für Clusteraufbau
- BI Integration



# Batch vs Echtzeit

Mit der **zunehmenden** Verbreitung von Batch basierten Lösungen, z. B. aufbauend auf **MapReduce**, fangen Anwender an **zeitnahe** Antwortzeiten zu verlangen. Es wird immer **wichtiger** den Batchbetrieb um „**Echtzeit**“ Komponenten zu **erweitern**.

Dabei gibt es dort **Ansätze** die entweder Daten **zuerst** Speichern oder aber **direkt** im Speicher verarbeiten.



# Gespeicherte Daten

Diese Verfahren haben wir **bereits** besprochen, wie zum Beispiel **Impala**. Die Daten werden **zuerst** abgespeichert und dann im **Nachlauf** verarbeiten. Dabei entsteht eine **Verzögerung** welche die **Aktualität** der Daten **beeinflusst**.

Der **Vorteil** dieser Lösungen liegt in der **Persistenz** und **Skalierbarkeit**.



# Hauptspeicher Daten

Neben der Verarbeitung der **gespeicherten** Daten gibt es auch Systeme, welche den ganzen Datenbestand im **Hauptspeicher** halten und dort **schnell** verarbeiten, z. B. SAP **Hana** oder Oracle **Exalytics**.

**Jedoch** sind diese typischerweise sehr **teuer**, denn RAM kostet **wesentlich** mehr als andere Speichertechnologien. Auch die **Skalierbarkeit** ist **fragwürdig**.



# Hauptspeicher Daten

Ein weiteres Thema ist **CEP**, das Complex Event Processing oder Stream Processing. Hier werden nur **bestimmte** Merkmale der Datensätze vorgehalten und zur **Abfrage** bereit gehalten, z. B. **Trending Topics**, oder **Summen** über Zeitfenster.

Ein Beispiel hier ist Apache **Storm** von Twitter (vorher BackType). Auch **Flume** kann **einige** Funktionen abdecken, ist aber generell **anders** konzeptioniert (Event Collection).

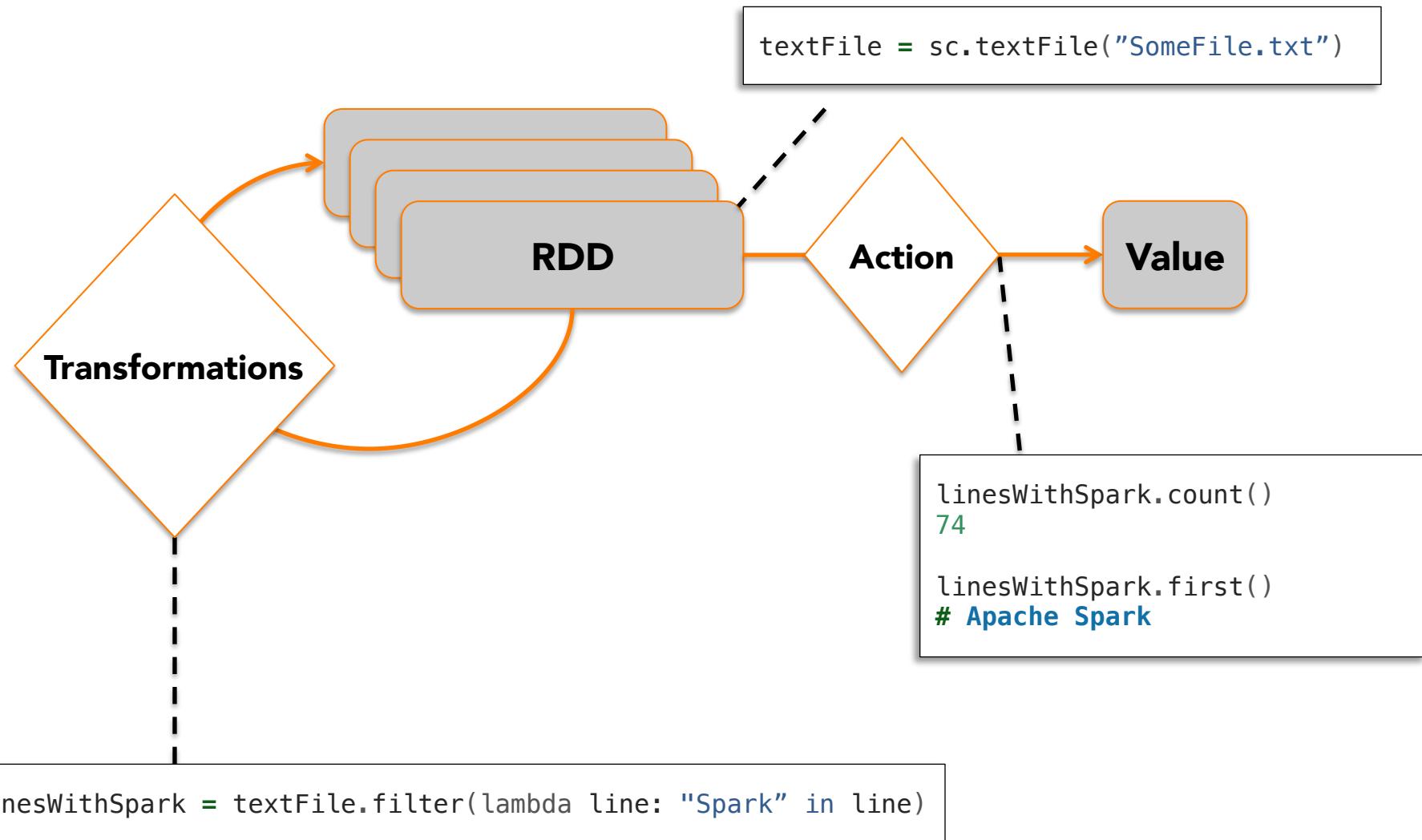


# Mischsysteme: Spark

Neben den **eindeutigen** System gibt es auch **Mischsysteme**, beispielweise **Spark**. Es basiert auf der Idee, Daten aus einem **Festspeicher** in den **Hauptspeicher** zu laden und dann für **iterative** Verarbeitung bereitzustellen (RDDs).

Spark hat zusätzlich ein **flexibleres** Datenverarbeitungsmodell. Es besitzt ein **Vokabular** welches es erlaubt die Schritte als Direct Acyclic Graphs (DAGs) zu definieren.

# Spark RDDs





# Mischsysteme: Spark

Man kann die Ablaufdefinitionen in **Java**, **Python** oder **Scala** schreiben.

**Ähnlich** der Sprache von Apache **Pig** gibt es **Transformationen** und **Aktionen**. Nur die letzteren stoßen die **eigentliche** Verarbeitung an und führen die bis **dahin** genannten Transformation aus.

Zwischendaten können **gecachet** werden damit **mehrfacher** Zugriff beschleunigt wird.



# RDDs Erstellen

```
# Turn a Python collection into an RDD
> sc.parallelize([1, 2, 3])
```

```
# Load text file from local FS, HDFS, or s3
> sc.textFile("file.txt")
> sc.textFile("directory/*.txt")
> sc.textFile("hdfs://namenode:9000/path/file")
```

```
# Use existing Hadoop InputFormat (Java/Scala only)
> sc.hadoopFile(keyClass, valClass, inputFmt, conf)
```



# Einfache Transformationen

```
> nums = sc.parallelize([1, 2, 3])  
  
# Pass each element through a function  
> squares = nums.map(lambda x: x*x)    // {1, 4, 9}  
  
# Keep elements passing a predicate  
> even = squares.filter(lambda x: x % 2 == 0) // {4}  
  
# Map each element to zero or more others  
> nums.flatMap(lambda x: range(x))  
  > # => {0, 0, 1, 0, 1, 2}
```



# Einfache Aktionen

```
> nums = sc.parallelize([1, 2, 3])
# Retrieve RDD contents as a local collection
> nums.collect() # => [1, 2, 3]

# Return first K elements
> nums.take(2) # => [1, 2]

# Count number of elements
> nums.count() # => 3

# Merge elements with an associative function
> nums.reduce(lambda x, y: x + y) # => 6

# Write elements to a text file
> nums.saveAsTextFile("hdfs://file.txt")
```

# Spark Beispiel: PageRank



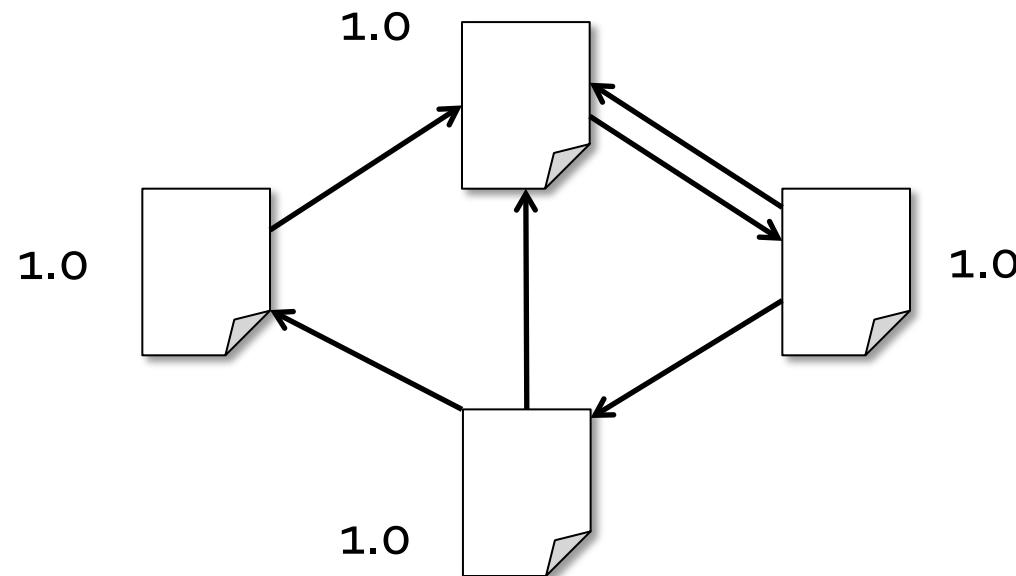
Spark **eignet** sich sehr gut für mathematische Verfahren, welche **Iterationen** benötigen, zum Beispiel der **PageRank** den Google bekannt machte. Dabei **wiederholt** sich die gleiche Berechnung so oft bis sich die Werte **nicht** weiter ändern. **Idee** ist:

- Links von **vielen** Seiten -> **Hoher Rang**
- Link von einer **hochrangigen** Seite -> **Hoher Rang**



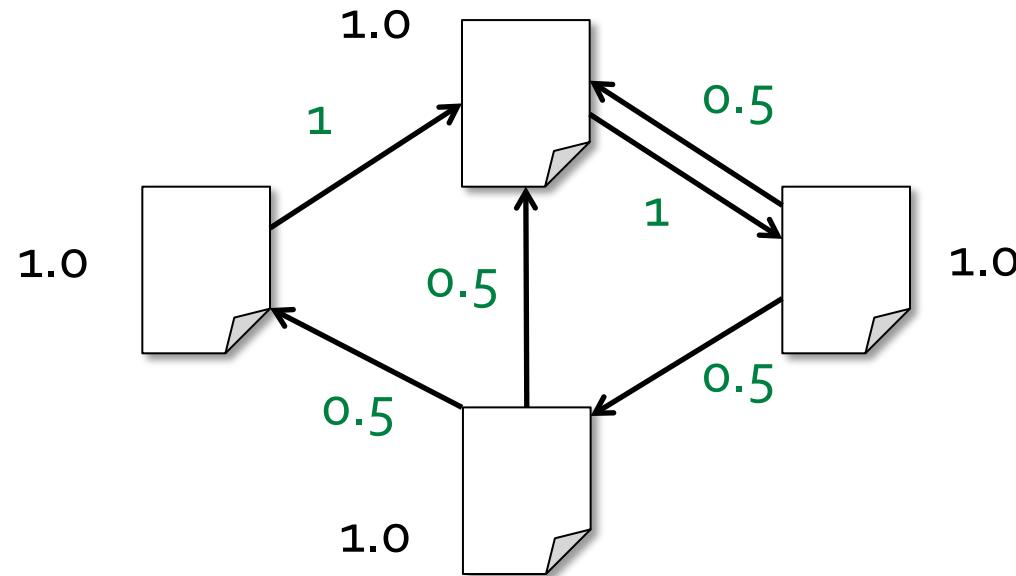
# PageRank Algorithmus

1. Jede Seite fängt mit einem Rang von 1 an
2. Bei jeder Iteration gibt Seite p einen Wert von  $\text{rank}_p / |\text{neighbors}_p|$  an Nachbarn ab
3. Setze Rang jeder Seite auf  $0.15 + 0.85 \times \text{contribs}$



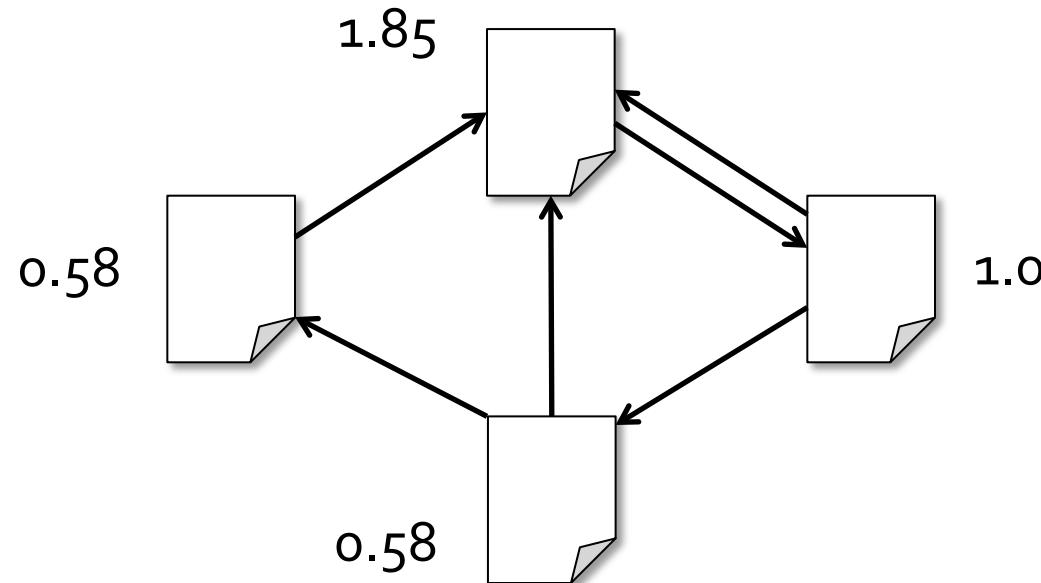
# PageRank Algorithmus

1. Jede Seite fängt mit einem Rang von 1 an
2. Bei jeder Iteration gibt Seite p einen Wert von  $\text{rank}_p / |\text{neighbors}_p|$  an Nachbarn ab
3. Setze Rang jeder Seite auf  $0.15 + 0.85 \times \text{contribs}$



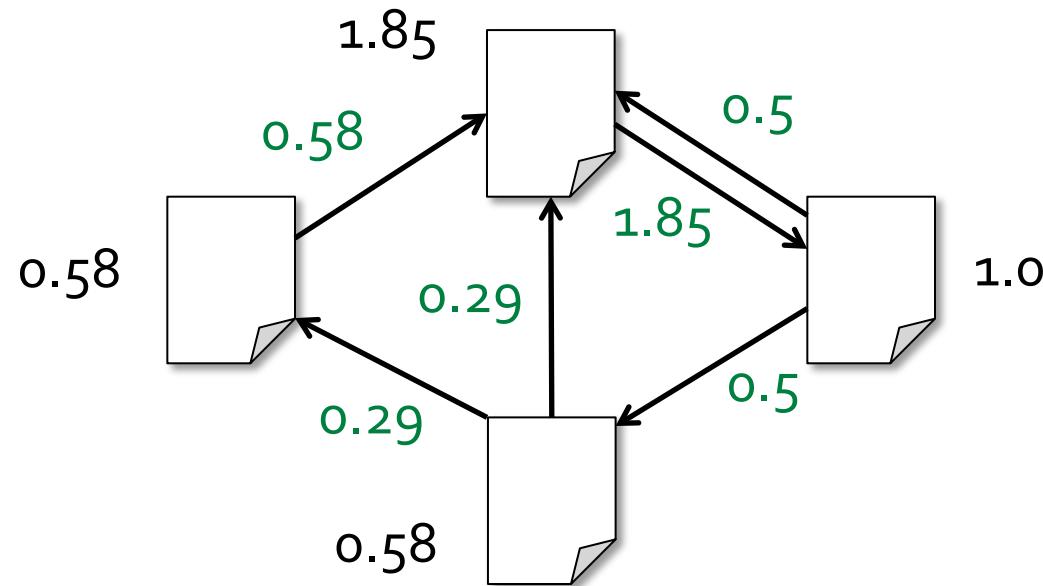
# PageRank Algorithmus

1. Jede Seite fängt mit einem Rang von 1 an
2. Bei jeder Iteration gibt Seite p einen Wert von  $\text{rank}_p / |\text{neighbors}_p|$  an Nachbarn ab
3. Setze Rang jeder Seite auf  $0.15 + 0.85 \times \text{contribs}$



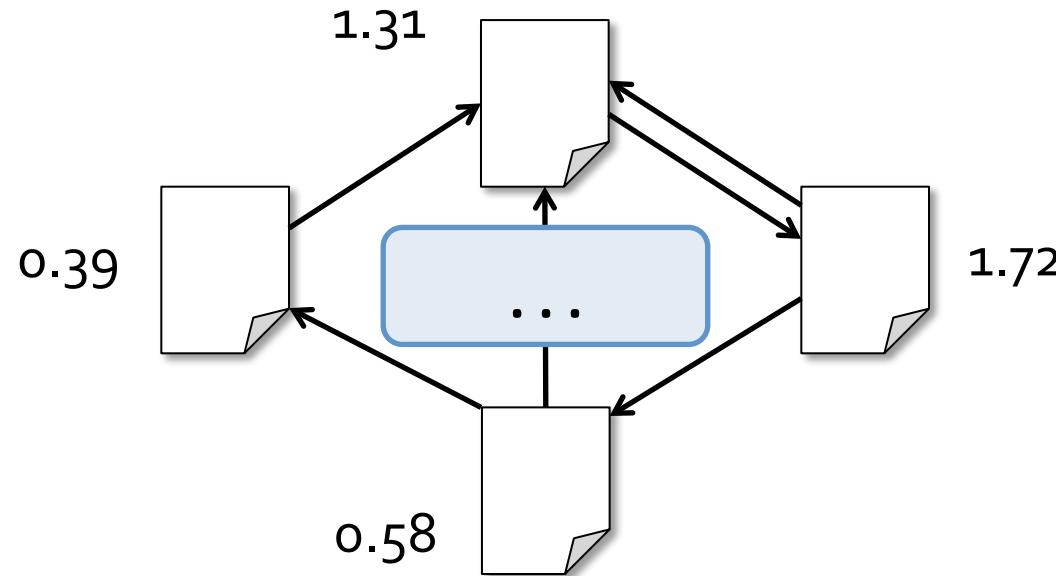
# PageRank Algorithmus

1. Jede Seite fängt mit einem Rang von 1 an
2. Bei jeder Iteration gibt Seite p einen Wert von  $\text{rank}_p / |\text{neighbors}_p|$  an Nachbarn ab
3. Setze Rang jeder Seite auf  $0.15 + 0.85 \times \text{contribs}$



# PageRank Algorithmus

1. Jede Seite fängt mit einem Rang von 1 an
2. Bei jeder Iteration gibt Seite p einen Wert von  $\text{rank}_p / |\text{neighbors}_p|$  an Nachbarn ab
3. Setze Rang jeder Seite auf  $0.15 + 0.85 \times \text{contribs}$

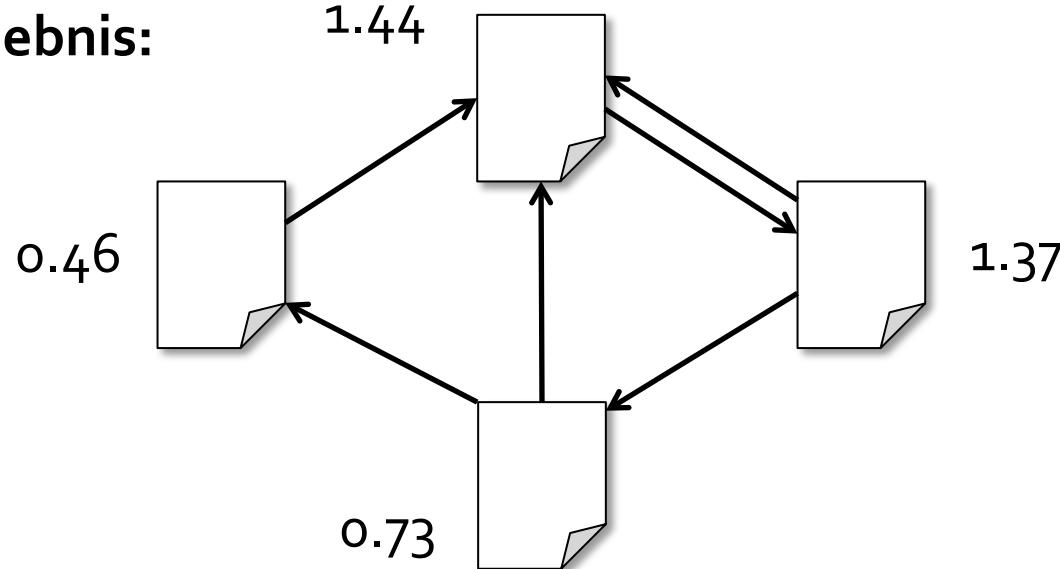




# PageRank Algorithmus

1. Jede Seite fängt mit einem Rang von 1 an
2. Bei jeder Iteration gibt Seite p einen Wert von  $\text{rank}_p / |\text{neighbors}_p|$  an Nachbarn ab
3. Setze Rang jeder Seite auf  $0.15 + 0.85 \times \text{contribs}$

Endergebnis:



# Spark Scala Implementierung

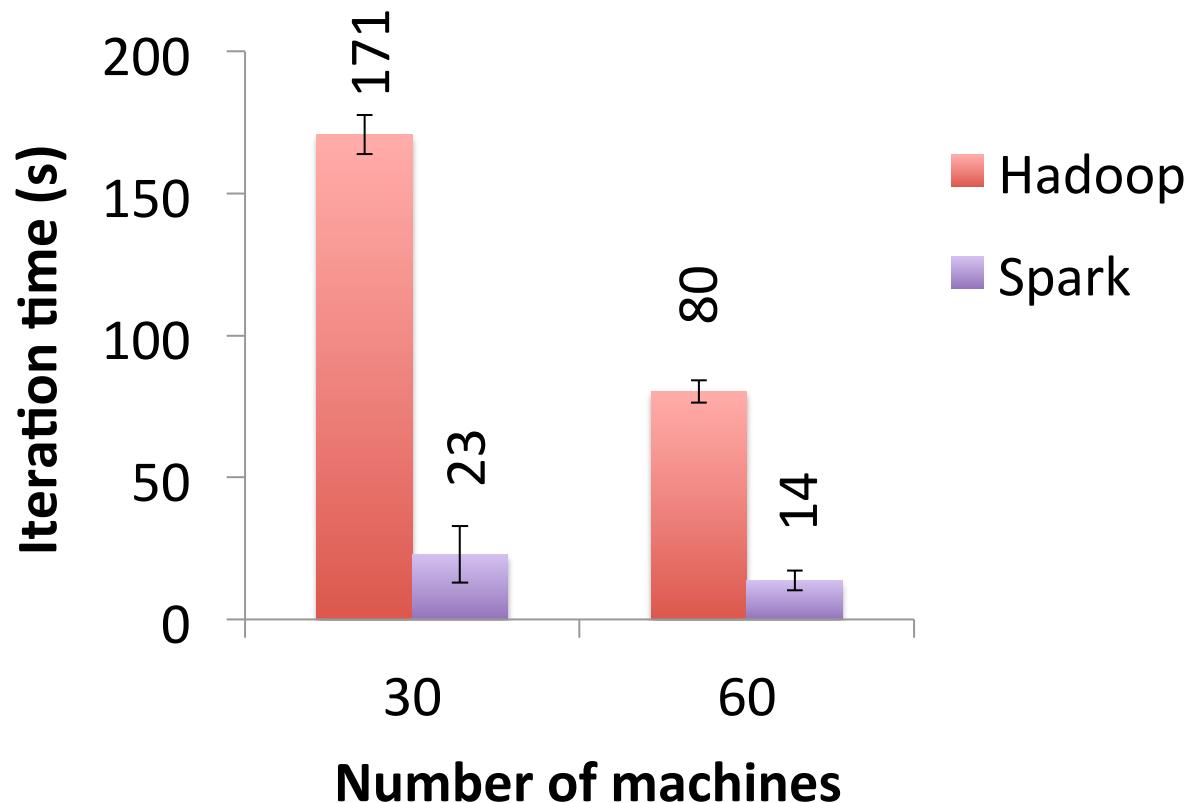


```
val links = // Load RDD of (url, neighbors) pairs
var ranks = // Load RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
    val contribs = links.join(ranks).flatMap {
        case (url, (links, rank)) =>
            links.map(dest => (dest, rank/links.size))
    }
    ranks = contribs.reduceByKey(_ + _)
        .mapvalues(0.15 + 0.85 * _)
}
ranks.saveAsTextFile(...)
```



# Spark PageRank Leistung





# Mischsysteme: Spark

Wie gezeigt kann Spark nicht nur **iterative**, sondern auch Algorithmen wie **MapReduce** abbilden. In der Praxis sind die **Laufzeiten** meistens um den Faktor **40-100** besser als bei dem **traditionellen** MapReduce.

Zusätzlich kann Spark auch **Stream Processing!** Spark ist nur **ein** Beispiel für wie sich die Datenverarbeitung in Hadoop **weiterentwickelt**.



# Datenaufnahme

Für ein System welches in „**Echtzeit**“ (soll hier heißen: **schnell genug** für den Menschen der auf eine Abfrageantwort wartet) arbeitet, aber nur **gespeicherten** Daten verarbeiten will/muss, ist eine **zeitnahe** Aufnahme der Daten für die weitere Verarbeitung unabdingbar.

Wie bereits erwähnt bietet Apache **Flume** eine Möglichkeit die Daten in **kleineren** Portionen zu verarbeiten, sogenannte **Ereignisdatenflüsse**.



# Flume

Flume bietet

- eine **skalierbare** Methode für das **Sammeln** und der **Aggregation** von Ereignisdatensätzen (z. B. Log Einträge)
- **dynamische**, kontextabhängige Ereignis **Weiterleitung**
- **niedrige** Latenz und **hoher** Durchsatz
- **deklarative** Konfiguration
- **sofortige** Einsatzfähigkeit, ist aber trotzdem **erweiterbar**

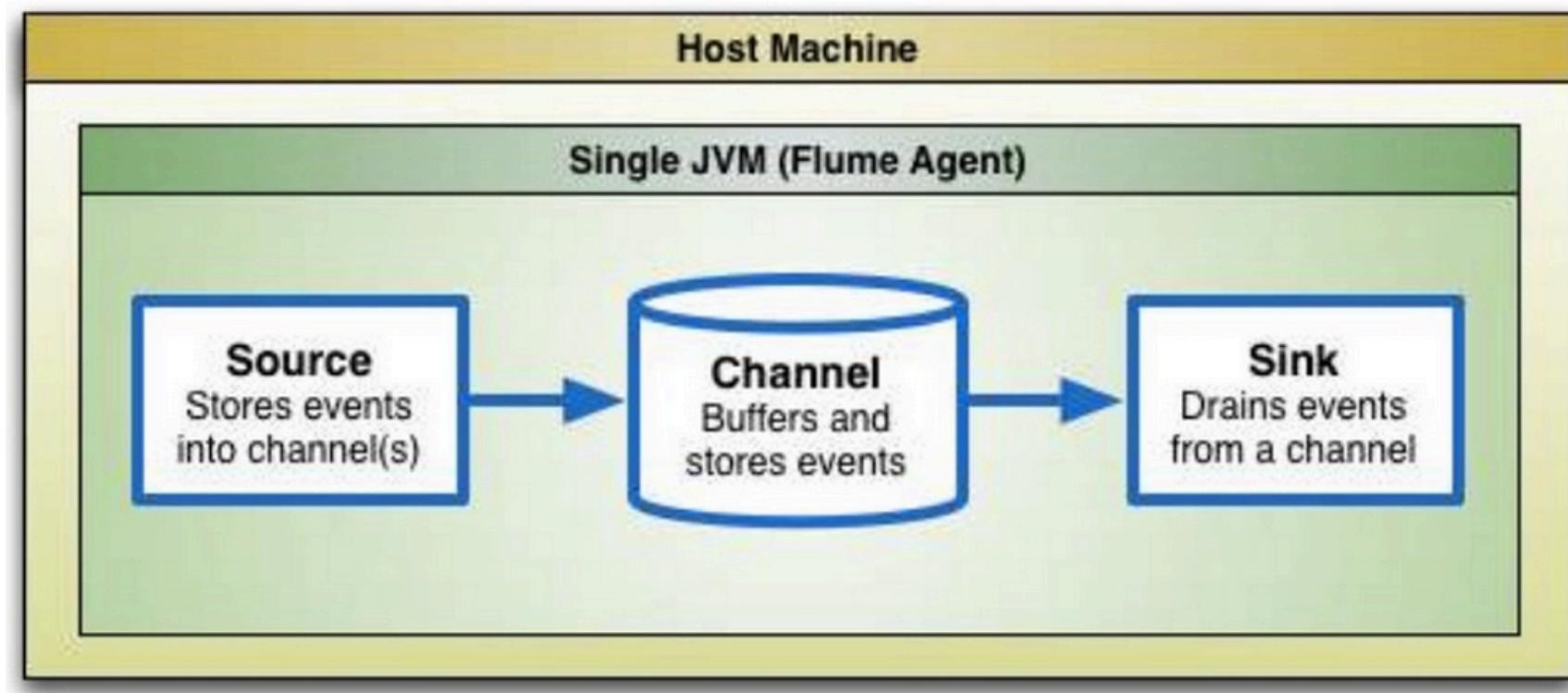


# Flume Agent

Flume setzt sogenannte **Agents** ein. Dieses sind **dedizierte** Prozesse (in Java) welche aus **mehreren** internen Komponenten bestehen. Diese sind die Quelle (**Source**), der Kanal (**Channel**) und der Abfluss (**Sink**). Sie sind innerhalb eines Agents **frei** definierbar und eine Menge an **vorgegebenen** Implementierungen werden bereits **mitgeliefert**. Agents werden dann **extern** zu einer Topologie zusammengeschlossen.



# Flume Agent





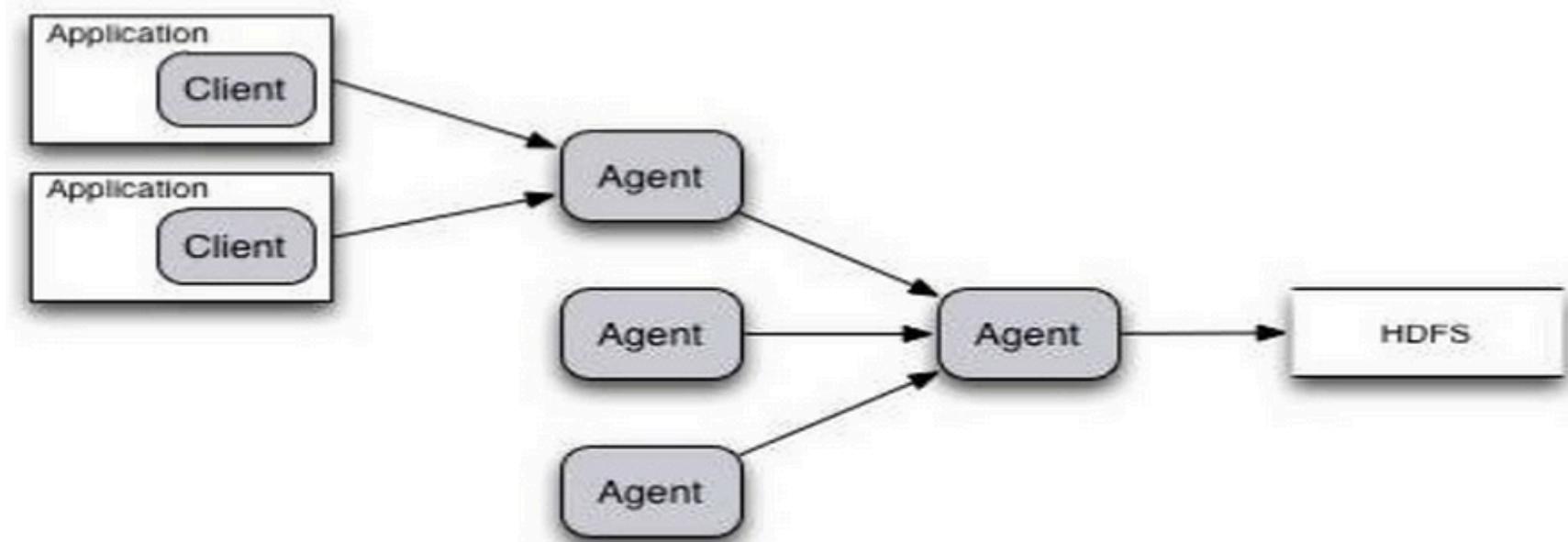
# Flume Topologie

In Flume können die Agents in eine **Topologie** zusammengefasst werden, welche **einfache**, aber auch **komplexe** Lösungen ermöglichen.

Der Sinn ist eine **verlässliche** Infrastruktur aufzubauen, die **ausreichend** dimensioniert (Parallelität für den Datendurchsatz und die Latenz) und gleichzeitig **ausfallsicher** ist. Dazu kann ein Datenfluss auch **aufgeteilt** oder **aggregiert** werden.



# Flume Topologie



$[\text{Client}]^+ \rightarrow \text{Agent} [\rightarrow \text{Agent}]^* \rightarrow \text{Destination}$



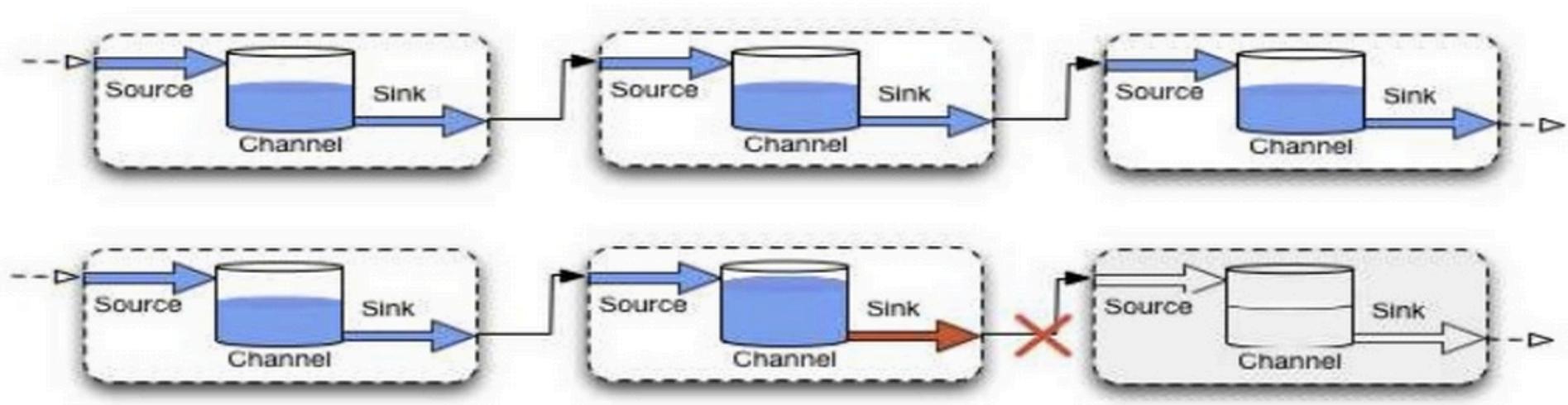
# Flume Hinweise

Der Zugriff auf eine Flume Agent läuft entweder über die native Java API oder mitgelieferte Quellen, wie log4j, syslog und so weiter.

Es muss mindestens einen Kanal geben und mindestens ein Quelle oder Abfluss. Darüber hinaus kann es aber beliebig viele Quellen, Kanäle und Abflüsse geben.



# Flume Konzepte

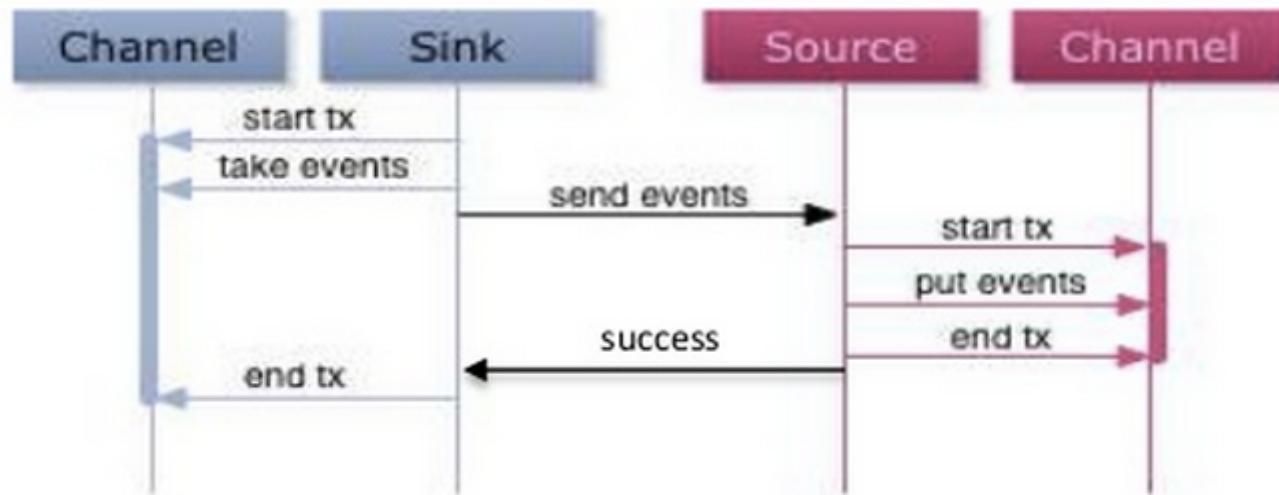


Die Quelle fügt Ereignisse dem Kanal hinzu. Der Abfluss nimmt Ereignisse aus dem Kanal heraus. Der Kanal speichert Ereignisse bis diese herausgenommen werden.



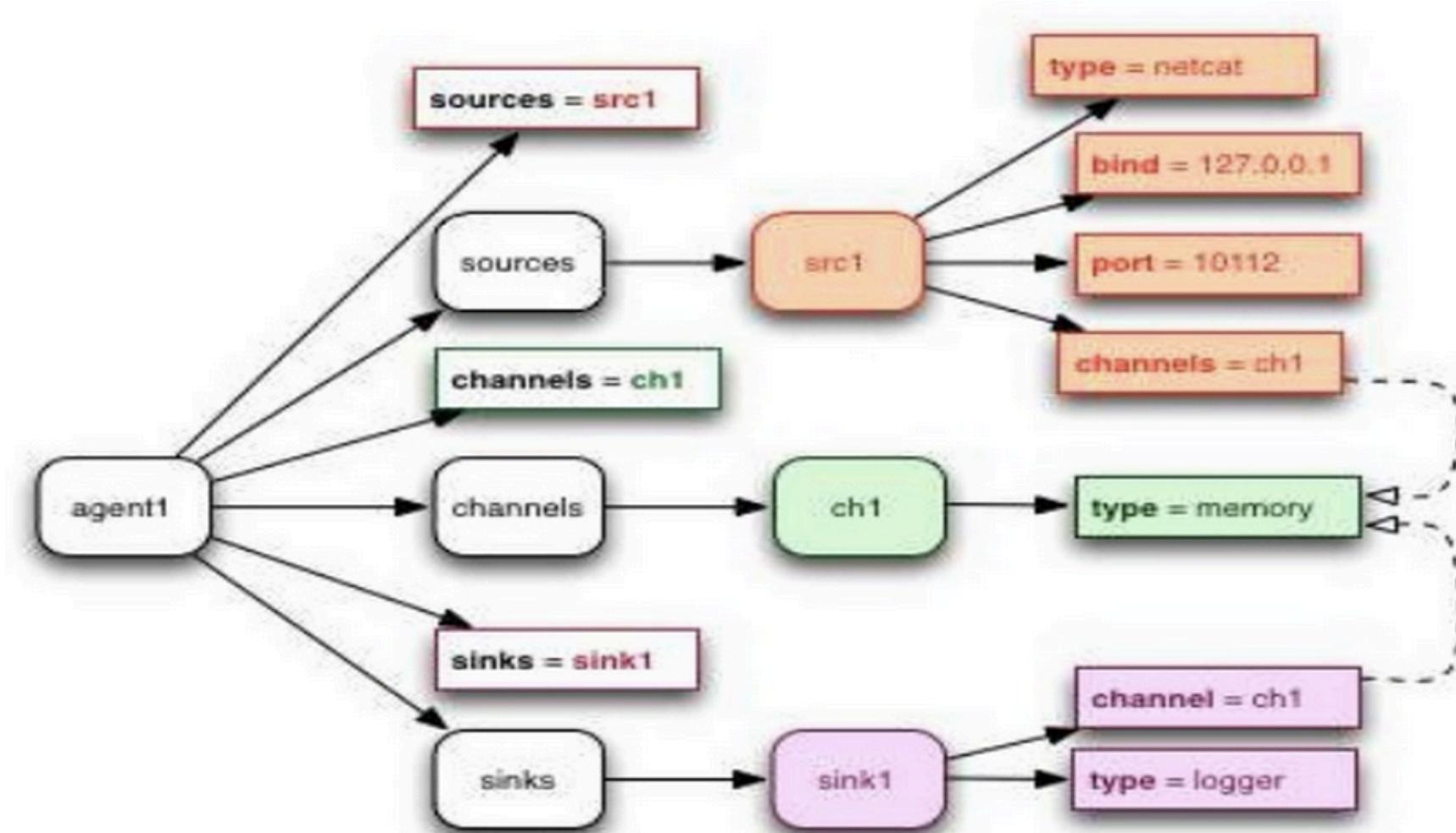
# Flume Verlässlichkeit

Die Kommunikation zwischen Agents ist transaktional. Dazu kommen die Persistenzeigenschaften des Kanals und formen zusammen die Verlässlichkeit der Nachrichtenübertragung.





# Konfigurationsbaum





# Agent Quellen

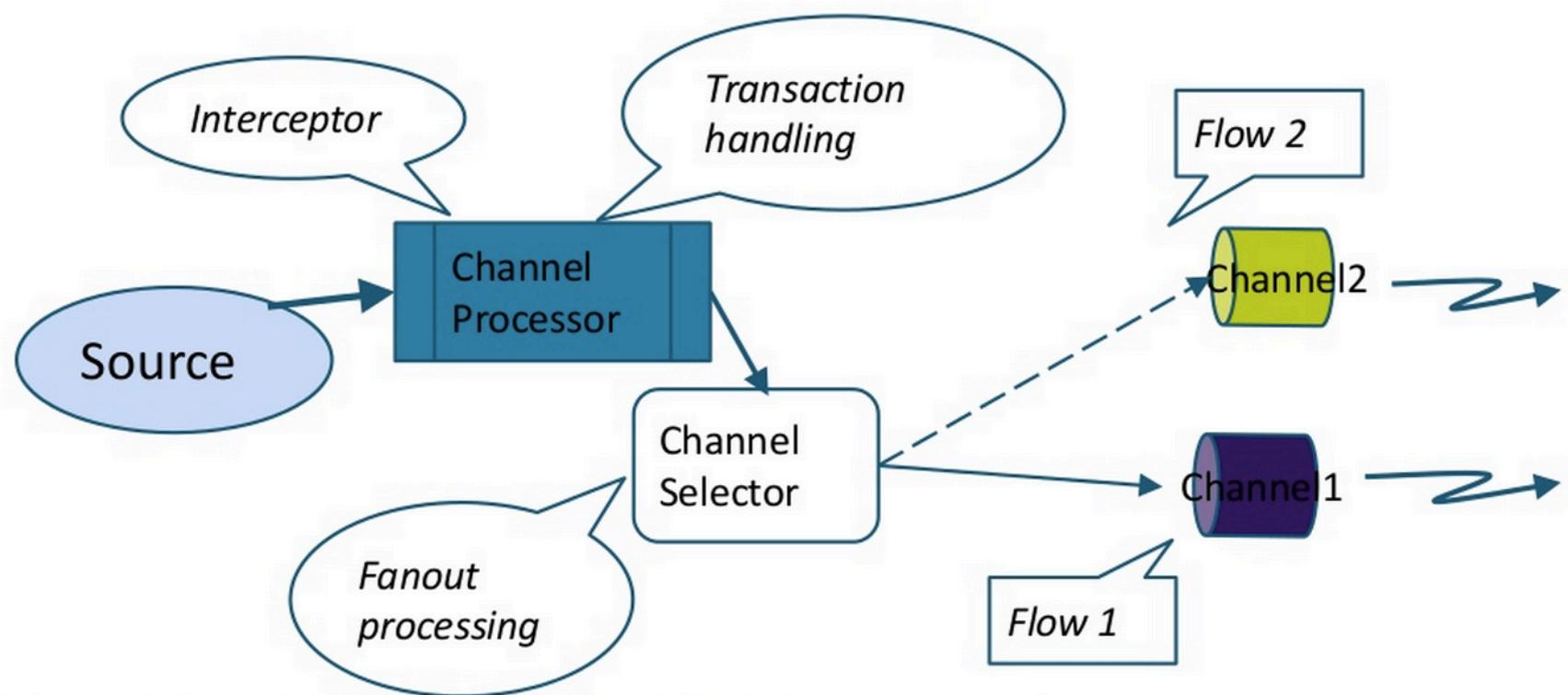
Die Quellen sind **ereignisgesteuert**, unterstützen aber auch die Verarbeitung von mehreren Ereignissen auf einmal (**Batch Processing**).

Es gibt:

- **AVRO** – RPC Schnittstelle (Agent an Agent)
- **Thrift** – RPC Schnittstelle
- **Spooldir** – Zum lesen von rotierenden Logdateien
- **HTTP** – REST API
- **JMS** – Java Message Service
- **SyslogTcp, SyslogUdp** – Linux/Unix Daemon
- **Netcat** – Socket Kommunikation
- **Exec** – Externe Skripte



# Aufteilung des Flusses





# Kanalauswahl (Selektoren)

Die Aufteilung kann für **alle** Ereignisse gelten (**Replicating** Selector, d.h. alle Ereignisse an alle Kanäle) oder nur für **bestimmte** (**Multiplexing** Selector, d.h. kontextabhängiges Weiterleiten).

Beispiel:

```
agent1.sources.sr1.selector.type = multiplexing
agent1.sources.sr1.selector.mapping.foo = channel1
agent1.sources.sr1.selector.mapping.bar = channel2
agent1.sources.sr1.selector.default = channel1
agent1.sources.sr1.selector.header = yourHeader
```



# Agent Kanäle

Der Kanal ist eine **passive** Komponente, ist aber entscheidend für die **Verlässlichkeit** des Datenflusses. Flume liefert einige Kanäle bereits mit:

- **File** – Speichert auf Datenträger zwischen
- **Memory** – Reine Hauptspeicher Variante
- **JDBC** – Nutzt eine Datenbank



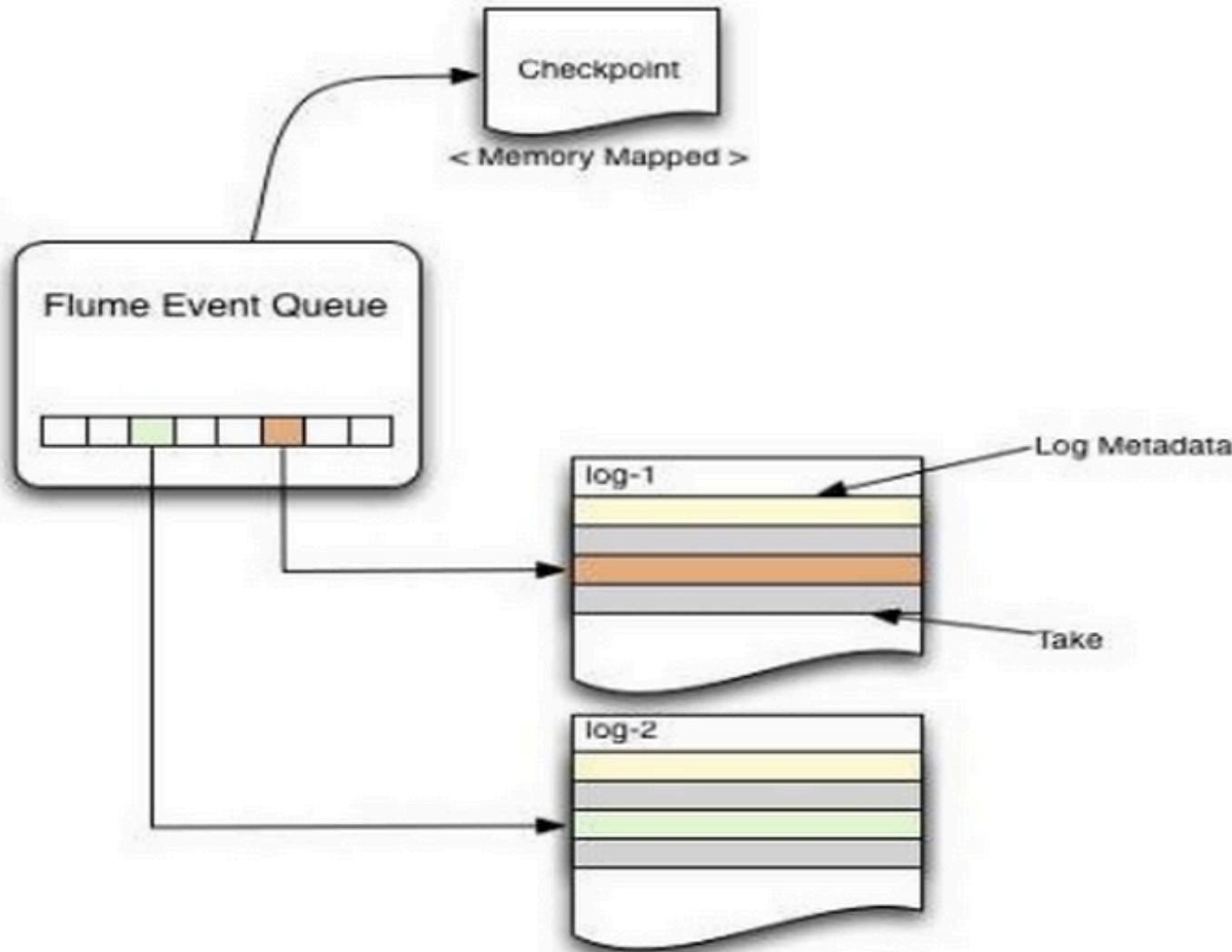
# Beispiel: File Channel

Der File Channel benutzt eine Write-Ahead Log Implementierung. Beispiel:

```
agent1.channels.ch1.type = FILE
agent1.channels.ch1.checkpointDir = <dir>
agent1.channels.ch1.dataDirs = <dir1> <dir2>...
agent1.channels.ch1.capacity = N (100k)
agent1.channels.ch1.transactionCapacity = n
agent1.channels.ch1.checkpointInterval = n (30000)
agent1.channels.ch1.maxFileSize = N (1.52G)
agent1.channels.ch1.write-timeout = n (10s)
agent1.channels.ch1.checkpoint-timeout = n (600s)
```



# Beispiel: File Channel





# Agent Abflüsse

Der Flume Agent **Abfluss** ist eine abfragende (**polling**) Komponente, welche Ereignisse aus dem **zugeordneten** Kanal abholt. Dies kann auch mit **mehrfachen** Ereignissen in einem Aufruf passieren (**batch**).

Mitgelieferte Abflüsse sind:

- **HDFS** – Schreibt in HDFS (vielfach konfigurierbar)
- **HBase** – Speichert Daten in HBase
- **AVRO, Thrift** – RPC Abfluss um Daten weiterzureichen
- **FileRoll** – Schreibt lokal in rotierende Dateien
- **Null, Logger** – Zum Testen
- **ElasticSearch** – Indiziert Ereignisse mit ES
- **IRC** – Zur Ausgabe im IRC



# Serialisierung und Gruppen

Für **bestimmte** Formate, bei denen kein **explizites** Format angegeben ist, gibt es eine **Serialisierungsschnittstelle**. Damit kann ein Ereignis in ein **beliebiges**, dem Anwender **passendes** Format umgewandelt werden.

Des Weiteren können Abflüsse in **Gruppen** eingeteilt werden (eine pro Abfluss) welche **zusätzliche** Eigenschaften global **definieren**, z. B. **Abfluss Prozessoren**.



# Anwendungsschnittstelle

Die Flume **Client** SDK **unterstützt** bereits das Umschalten bei fehlerhaften Verbindungen (**failover**) und das **Verteilen** der Last über **Round Robin**, **Zufall** oder **eigenen** Selektoren. Außerdem wird **Thrift** und **Avro** unterstützt.

**Factory** Code:

```
org.apache.flume.api.RpcClientFactory:  
    RpcClient getInstance(Properties)  
org.apache.flume.api.RpcClient:  
    void append(Event)  
    void appendBatch(List<Event>)  
    boolean isActive()
```



# Weitere Konzepte

Flume hat auch eine einfache **Datenverarbeitungskomponente**, die **Interceptors**. Diese erlauben es die Datensätze während sie im Agenten gehandhabt werden zu verändern. Man kann hier **interessante** Eigenschaften **extrahieren** und z. B. der Nachricht als **Kopfinformation** hinzufügen. Das kann im **Selektor** wiederum für das selektive Weiterleiten genutzt werden.



# Generelle Hinweise

**Verlässlichkeit** ist ein Faktor des **Kanaltyps**,  
**Kapazität** und System **Redundanz**.

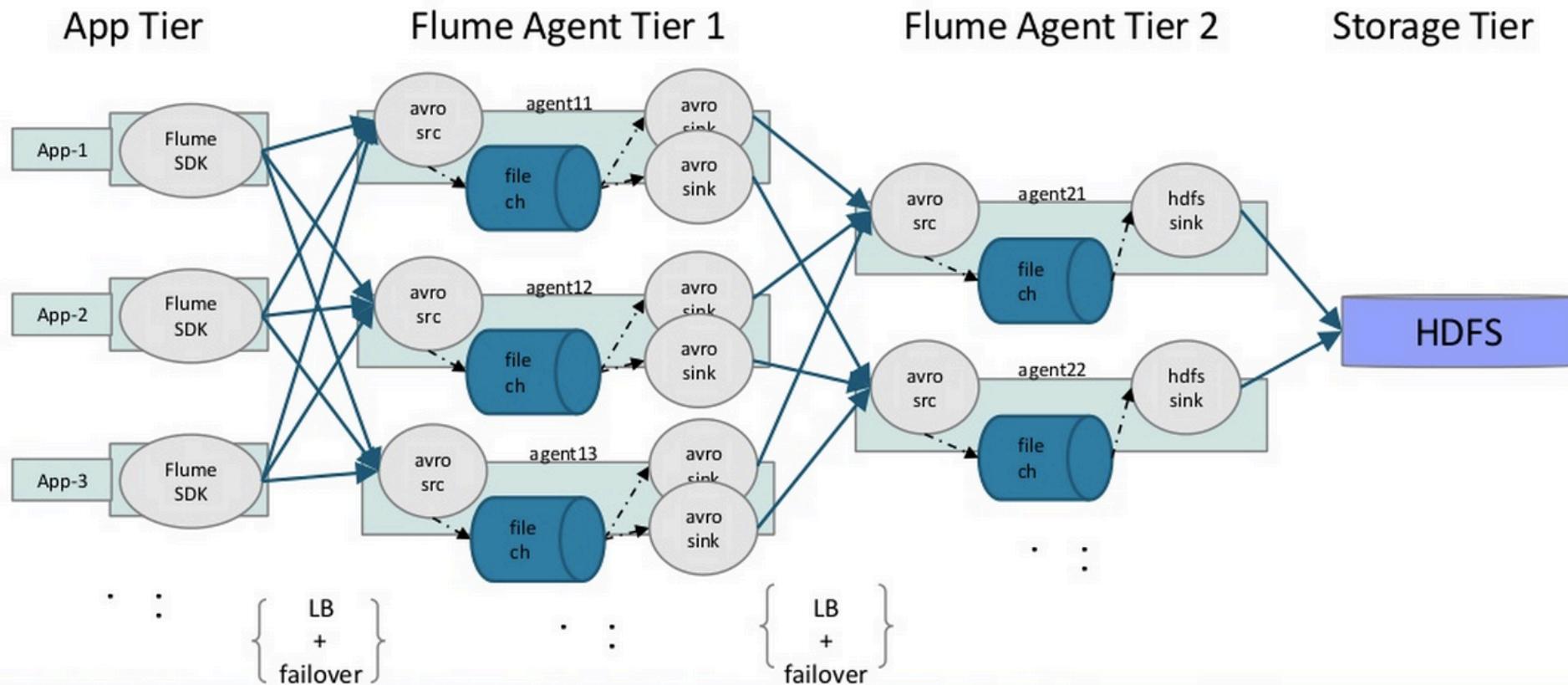
Die Kanal **Größe** ist wichtig, um der Aufgabe  
**gerecht** zu werden (Alles Physik!)

Die **Batchgröße** sollte an die Anwendung  
**angepasst** sein.

Anzahl der CPU **Cores** sollte die **Hälfte** der  
gesamten Anzahl von **Quellen** und **Abflüssen**  
innerhalb **eines** Agents sein (Threads!).



# Beispiel: Komplexe Topologie





# Datenaufnahme

Damit **beenden** wir die Thematik der Datenaufnahme.

In der **Kombination** der Aufnahme mit der Verarbeitung liegt der **essentielle** Kern des Big Data Engineerings, wenn es **speziell** um die Ingenieursaufgabe geht.

Neben der **schwierigen** Aufgabe des **Verständnis** der Daten muss ein BDE auch noch die beste **physikalische** Lösung implementieren.



# Einheit 7

- Rückblick auf Einheit 6
- Hybride Architekturen
  - Batch und Real-time gemischt
- „**Best Practices**“ für **Clusteraufbau**
- BI Integration



# Clusteraufbau

Beim **Clusteraufbau** muss auf den **Anwendungsfall** geachtet werden. Es gibt solche Fälle wo möglichst **viele** Daten gespeichert werden sollen, oder solche wo ein möglichst **hoher** Datendurchsatz erreicht werden soll.

Hier **variieren** die **Anzahl** und **Art** der Speicherplatten, sowie die **Größe** des Hauptspeichers. Im folgenden sind einige **Beispiele** genannt.



# Clusteraufbau

Beispiel #1: Gemischter Einsatz, d. h.  
MapReduce, Spark, Impala, HBase etc.

CPU	2 x Quad Core CPU
RAM	48-96GB
Platten	6-12 x 1-2TB HDDs (SATA)
Netzwerk	1 oder 10 GigE



# Clusteraufbau

Beispiel #2: Hohe Speichererdichte, z. B. für große Datenarchive

CPU	2 x Quad Core CPU
RAM	48-96GB
Platten	24-72 x 1-4TB HDDs (SATA)
Netzwerk	1 oder 10 GigE



# Clusteraufbau

Beispiel #3: Hohe Durchsatz, kleine Latenz, z. B.  
für Online Systeme

CPU	2 x 8 Core CPU
RAM	96-128GB
Platten	4 x 3-6TB SSDs (PCI)
Netzwerk	10 GigE oder InfiniBand



# Rack Awareness

Hadoop unterstützt das Konzept des **Rack Awareness**. Es hilft bei der **Verteilung** der Daten über solche Rechner, die **nicht** an dem **gleichen** Stromkreis und Netzwerkswitch **angeschlossen** sind. Beim Ausfall eines **ganzen** Stromkreises sind trotzdem **alle** Blöcke – wenn auch **verringert** – verfügbar.

In Hadoop wird dies über ein einfaches **Skript** gesteuert, welches an den Cluster angepasst werden muss.



# Netzwerktopologie

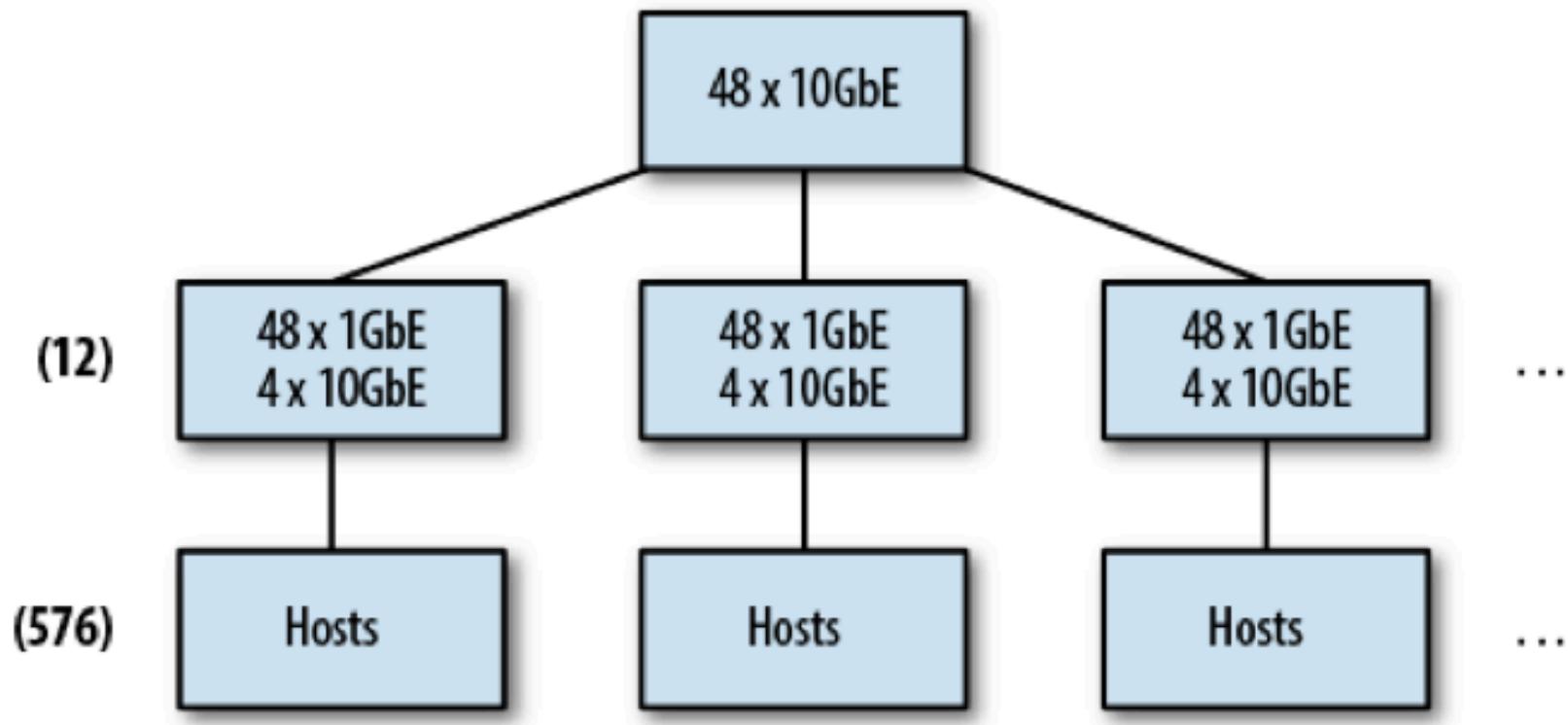
Ein weiterer **wichtiger** Faktor für eine gute Clusterleistung ist das **Netzwerk**. Es ist anzumerken, dass **viele** Netzwerk-Switches **nicht** die **angegebene** Leistung wirklich in der Praxis **liefern**. Hier zählt: **mehr** Leistung **kostet** mehr – oder: Man bekommt was man bezahlt.

Schnittstellen können über das **Bonding** weiter ausgebaut werden. Aber hier können auch **Probleme** entstehen. **Testen!**

# Netzwerktopologie

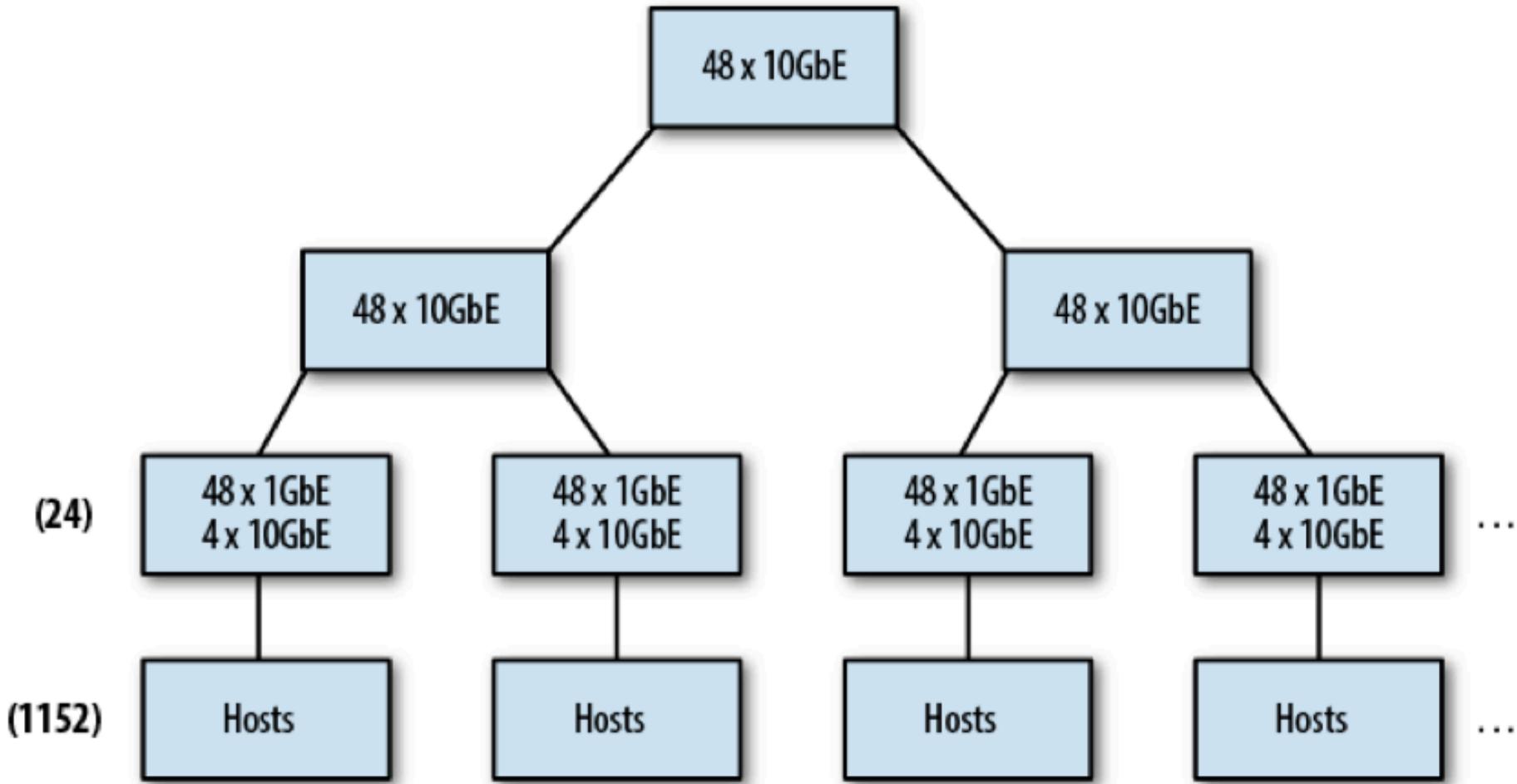


Beispiel: Baum, zwei Ebenen, 576 Rechner



# Netzwerktopologie

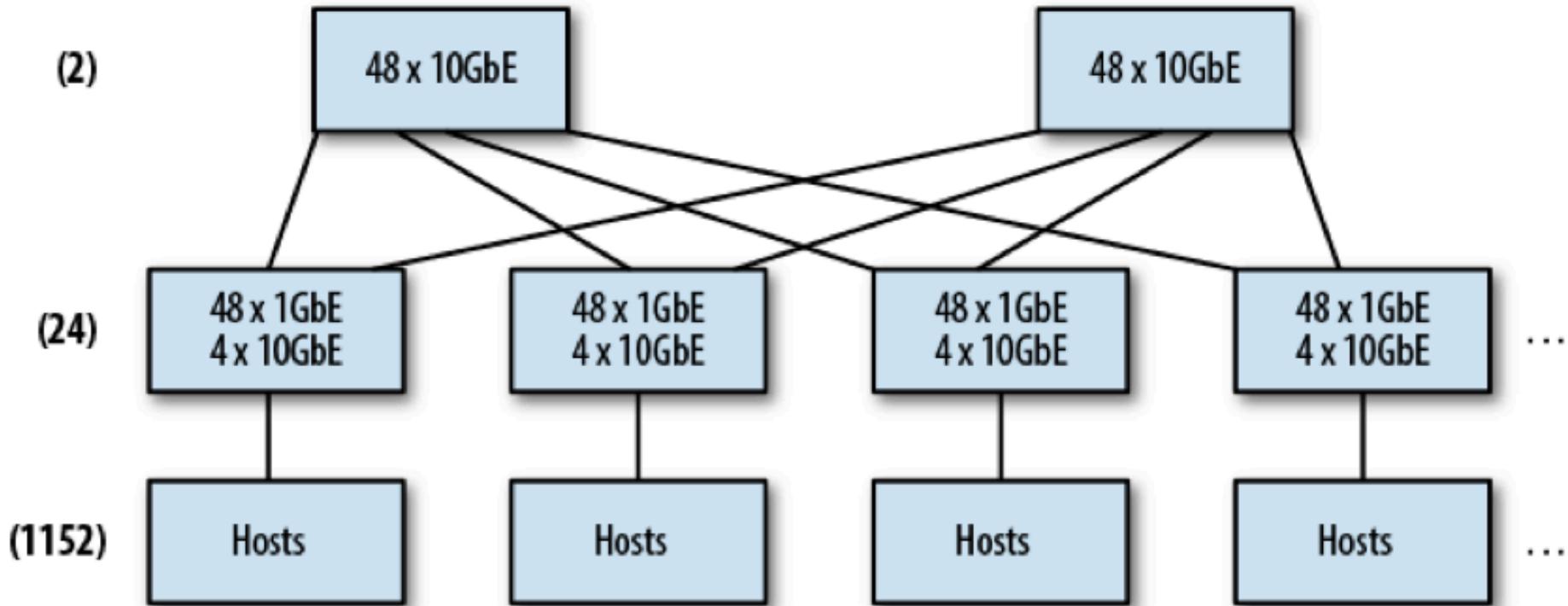
Beispiel: Baum, drei Ebenen, 1152 Rechner





# Netzwerktopologie

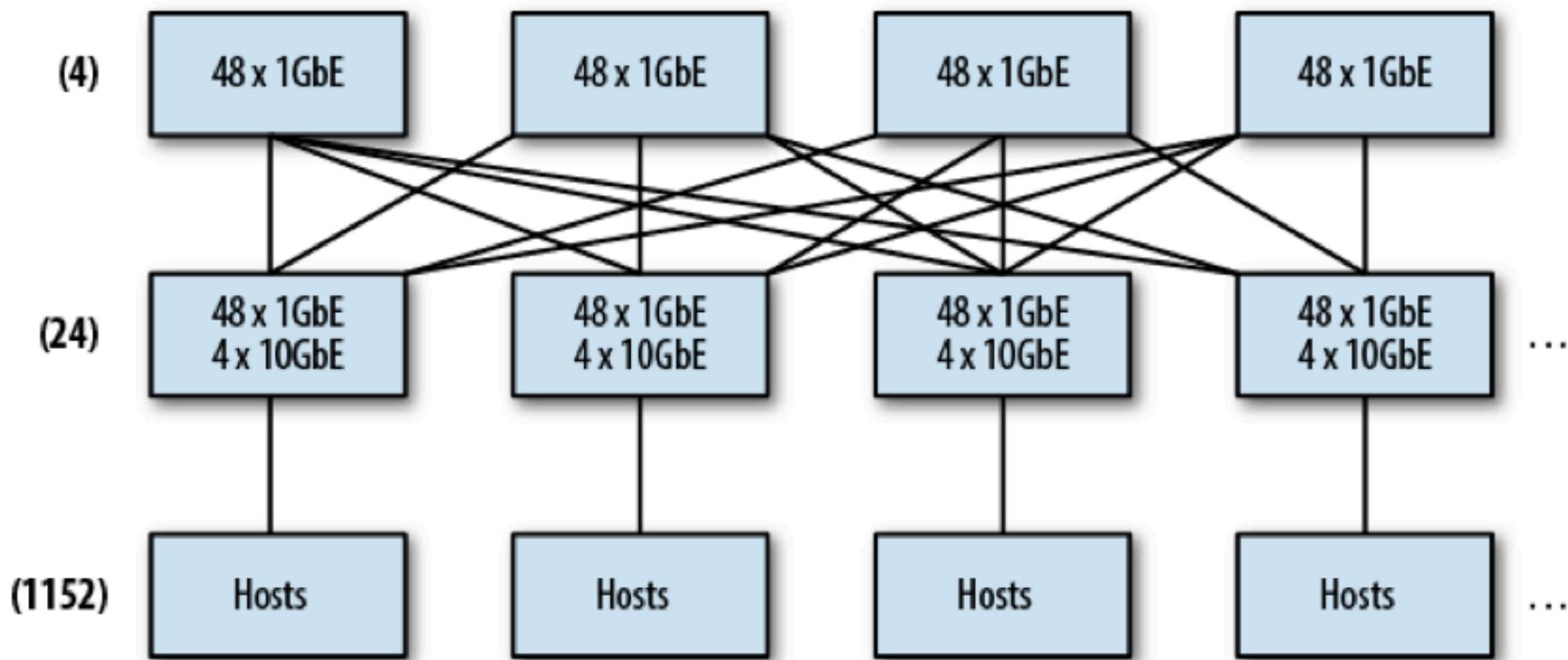
Beispiel: Spine, zwei Switches, 1152 Rechner





# Netzwerktopologie

Beispiel: Spine, vier Switches, 2304 Rechner





# Checkliste

1. Ist das Netzwerk so schnell wie angegeben und zeigt keine Fehler in allen Komponenten (Switch und Knoten)?
2. Sind die Rechner voll funktionsfähig in allen Komponenten (Daten- und Hauptspeicher)?
3. Ist die Software richtig installiert, d. h. ist alles voll funktionstüchtig/kompatibel?
4. Grundregel: Testen und beobachten!



# Einheit 7

- Rückblick auf Einheit 6
- Hybride Architekturen
  - Batch und Real-time gemischt
- „Best Practices“ für Clusteraufbau
- **BI Integration**



# BI Integration

Der **abschließende** Aspekt der Datenverarbeitung ist die **Darstellung** in Grafiken, Reports und/oder Dashboards.

Dazu werden die **Ergebnisse** mittels **Schnittstellen** an bestehende (oder auch **neue**, spezialisierte) Werkzeuge übergeben. Typisch hier sind die **Business Intelligence (BI)** Tools.

Diese **haben** bereits bestehende, etablierte **Schnittstellen**.



# JBDC/ODBC

Sehr oft werden Datenbank mittels **JBDC/ODBC** angesprochen, denn diese **Schnittstelle** ist **standardisiert** und bietet eine **gute** Grundlage.

**JBDC** ist die Java Implementierung, und **ODBC** die vergleichbare für den nicht-Java Welt.  
**Solange** eine Entwicklungssprache und -umgebung einen ODBC Treiber **bereitstellt**, **sollte** eine Verbindung mit den **gängigen** Datenbanksystemen **möglich** sein.



# JBDC/ODBC

Darauf zu **achten** ist aber, dass bei der **Anbindung** von Big Data Lösungen ggf. sich **bestimmte** Merkmale positiv oder negativ **auswirken** können.

Ein **Beispiel** ist die Latenzzeit bei der Benutzung von **Hive** durch JDBC. Hive braucht **mindestens 20-30** Sekunden pro Abfrage wegen der **Batchnatur** des Systems. **Impala** beispielsweise kann dieses Problem **lösen**.



# Anbindung

Wie **immer** im Big Data Engineering kommt es darauf an die Systeme **gut** genug zu verstehen, um die **passende** Kombination möglichst **optimal** auszuwählen. Dabei gibt es **manchmal** (eher **oft**) die Möglichkeit die Lösung für Teilaufgaben zu **optimieren**, während andere Bereiche bewusst etwas **schlechter** laufen.

Auch hier heißt es wieder: **Testen** und **beobachten!**



# Einheit 7

An dieser Stelle endet die siebte und letzte Einheit, welche abschließende Konzepte und Hinweise zusammenfasst.

Vielen Dank für die Teilnahme am Kurs, und viel Glück bei dem Erstellen von Big Data Lösungen.

Auf Wiedersehen!

Lars George



# Übung 7

## Ziele:

- Arbeit am Kursprojekt
- Arbeit an offenen Übungen



# Übung 7

**Code:**

<https://github.com/larsgeorge/fh-muenster-bde-lesson-7>



# Quellen

- Flume
  - Projekt: <http://flume.apache.org/>
  - Präsentation:  
<http://www.slideshare.net/ydn/flume-hug>
- „Hadoop Operations“
  - Eric Sammer, O'Reilly