

# Masterstudiengang Wirtschaftsinformatik

# Big Data Engineering

FH Münster  
Master Wirtschaftsinformatik  
Wintersemester 2013  
Dozent: Lars George



# Einheit 6

- Rückblick auf Einheit 5
- Diskussion: BDE Zwischenstand
- Datenpipelines
  - Oozie, Crunch, Cascading, Morphlines
- **Hauptziel:** Gelernte Techniken kombiniert zur Verarbeitung von Daten einsetzen. Robuste Datenpipelines aufbauen.
- **Übung 6:**
  - Oozie Workflow erstellen und schon bekannte Methoden dort kombinieren



# Einheit 6

- **Rückblick auf Einheit 5**
- Diskussion: BDE Zwischenstand
- Datenpipelines



# Rückblick auf Einheit 5

- Fragen?
  - HBase ausprobiert?
  - Andere NoSQL Datenbank ausprobiert?
  - Hush?
- Übung 5: HBase Tabellen?



# Einheit 6

- Rückblick auf Einheit 5
- **Diskussion: BDE Zwischenstand**
- Datenpipelines



# BDE Zwischenstand

Eine Frage...

**Was ist  
Big Data Engineering?**



# BDE Zwischenstand

Wir versuchen nun **zusammenzufassen**, was Big Data Engineering (**BDE**) genau **ist**, nachdem wir **viele** Teile des **Hadoop** Ökosystems angeschaut haben.

Ist BDE nicht das **Gleiche**, wie schon **normale** Datenverarbeitung zuvor? Was **bringt** denn BDE an **zusätzlichen** Vorteilen? Was bringt es für **Nachteile**, oder wo sind die **Schwierigkeiten** verborgen?



# Big Data Engineering

BDE ist ein **evolutionärer** Ansatz, welcher durch **Moore's** Gesetz erst möglich wurde: Durch die ewig **steigende** Computerleistung bei **gleichbleibenden** Kosten, kann man heute **kostengünstig** riesige Rechnerverbunde aufbauen, welche sich aus **einfach** wartbaren Komponenten zusammensetzt. Damit kann man Daten **speichern** – und **verarbeiten!** – welche in der Vergangenheit **ignoriert** wurden.





# Big Data Engineering

**Weiterhin** werden diese einfach aufgebauten Systeme auch mit dem Hinblick auf **Ausfall** konzipiert. Jede Hard- und Softwarekomponente kann **leicht** ausgetauscht werden.

Bei der Software bedeutet dies, dass Prozesse **redundant** und **hochverfügbar** aufgebaut sind, damit **niemals** ein Service ausfällt und die Leistung sich **linear** mit dem Umfang des Ausfalls **verringert**.



# Big Data Engineering

**Erste** Systeme im BDE Umfeld waren **reine** Batchsysteme, denn diese **erlauben** es riesige Datenmenge verteilt zu **speichern** und **verarbeiten**. Es wurde auf **Durchsatz** und weniger auf **Latenzzeiten** geachtet.

Mit dem **weiteren** Fortschritt der Hard- und Software kann man nun auch **interaktive** („Echtzeit“) Systeme verwirklichen, welche auf **derselben** Hardware laufen.



# Big Data Engineering

Durch **geschickte** Nutzung von Prinzipien der **verteilten** Systeme, wie zum Beispiel **Replikation** von Daten und der Speicherung in speziellen **Dateiformaten**, welche dem Anwendungsfall **angepasst** sind, kann selbst auf **großen** Datenmengen eine **zufriedenstellende** Leistung erbracht werden.

**BDE ist die Anwendung, aber auch Entwurf und Implementierung solcher Systeme!**



# Big Data Engineering

Ein **Big Data Engineer** sollte die Merkmale der verteilter Verarbeitung kennen, also die

- **Architektur** solcher Systeme,
- deren **Schnittstellen** (nativ oder REST etc.),
- **Eigenheiten** (Konsistenzgarantie, Transaktionen, Index, Durchsatz, Latenz) und
- **Vor- und Nachteile** (Datenverlust?).

Damit kann die richtige Entscheidung getroffen werden, für einen gegebenen Anwendungsfall.



# Big Data Engineering

Neben der **Auswahl** der Hardware und Softwareplattform, hat BDE aber eine weitere **Disziplin**, die der **Datenverarbeitung** selbst. Dazu gehört das **tiefe** Verständnis der Bedeutung der Daten. Hier kommt das **Data Science** ins Spiel. Es sind aber **nicht** nur die **mathematischen** Formeln wichtig, sondern auch wie die Daten **vorbereitet** werden. Im BDE Umfeld stammen Daten oft **nicht** aus eigenen Quellen und müssen **angepasst** werden.



# Big Data Engineering

**Semi- oder unstrukturierte** Daten bedeutet meistens nur „**nicht mein Schema**“. Wenn zum Beispiel **Twitter, Facebook, Wetter** und **Kursdaten** in einem CRM System korreliert mit den **Kundendaten** dargestellt werden sollen, dann müssen diese für die **aktuelle** Auswertung ausgelesen werden. Manchmal braucht es dafür sogar **eigene** Dateiparser.



# Big Data Engineering

Ein **weiterer** Aspekt der Vorbereitung ist auch die **Ablage** der Daten im **richtigen** Datenformat, welches **einfache** Dateien, **spezielle** Dateiformate, wie auch spezielle **Speichersysteme** (z. B. HBase) umfasst.

Wichtig ist auch die **Partitionierung** der Daten (wenn diese nicht implizit stattfindet), zum Beispiel in HDFS. Oder des **Schlüsselentwurfs** in HBase (Schemadesign).



# Big Data Engineering

Hat man sich **entschieden**, welche **Hard-** und **Softwarekomponenten**, sowie **Datenquellen** und **Algorithmen** eingesetzt werden sollen, bleibt zuletzt die **Übergabe** der Lösung in den **Produktionsbetrieb**. **Datenaufnahme**, -**verarbeitung** und -**ausgabe** sollten möglichst **automatisiert** werden. Dazu **landen** die Daten in einem **Auffangbereich** und **stoßen** die Verarbeitungskette(n) an.





# Big Data Engineering

**BDE** ist zusammenfassend die Aufgabe Daten **innovativ** und **umfassend** zu verarbeiten. Dazu gehört die Auswahl (oder Erstellung)

- der Hard- und Softwarekomponenten,
- der Datenquellen und Vorbereitungsschritte,
- geeigneter Verarbeitungsalgorithmen,
- der Ausgabeschnittstellen,
- sowie die Automatisierung der Prozesse für die Produktion.



# Einheit 6

- Rückblick auf Einheit 5
- Diskussion: BDE Zwischenstand
- **Datenpipelines**



# Datenpipelines

Wie besprochen **braucht** man für das Verarbeiten von Daten **mehrere** Schritte, die sich mit **Teilaufgaben** beschäftigen. Dazu gehört die **Bereinigung** von Daten, vor der **eigentlichen** Analyse bis hin zur Verbindung **mehrerer, einzelner** Verarbeitungsprozesse zu einem **komplexeren** Prozess.

Die **einzelnen** Schritte auf jeder dieser Ebenen können **wiederverwendet** werden.



# Datenpipelines

Die Datenpipelines **teilen** sich demnach in mindestens **zwei** Klassen auf: **Makro** und **Mikro** Pipelines.

Die **Makro** Pipelines werden von den **Prozessverwaltungssystemen** (Scheduler) bestimmt. Beispiele hier sind Apache **Oozie** oder **Azkaban**.

Die **Mikro** Pipelines sind **Programmierhilfen** für die eigentliche Verarbeitung der Daten, wie beispielsweise **Crunch**, **Cascading** oder **Morphlines**.



# Makro Pipelines

Die sogenannten **Scheduler** Systeme sind normalerweise **eigenständige** Prozesse, welche **komplexe** Datenverarbeitung durch das Verknüpfen von **einzelnen** Arbeitsabläufen ermöglichen.

Wir werden uns dies am Beispiel von **Oozie** anschauen. Ein anderes Beispiel ist **Azkaban**, entwickelt von LinkedIn (siehe <http://azkaban.github.io/azkaban2/>), oder **Luigi** von Spotify (<https://github.com/spotify/luigi>).



# Apache Oozie

Wie erwähnt ist es in der Praxis oft wichtig mehrere Teilaufgaben zu **kombinieren**. Apache **Oozie** ist ein serverbasiertes **Koordinations-system**, welches sich auf Abläufe (**Workflows**) spezialisiert. Diese Abläufe können über **Zeit-** oder **Datenauslöser** gestartet werden, z. B. „Jede Stunde“ oder „Wenn die Eingabedaten komplett vorliegen“.

Abläufe werden als **Graphen** spezifiziert.



# Oozie Abläufe

Oozie **Abläufe** sind eine Menge an **Aktionen** (Actions), z. B. MR, Hive/Pig Aufträge, welche in einem **kontrollabhängige DAG (Direct Acyclic Graph)** beschrieben sind. Kontrollabhängig für Aktionen **bedeutet**, dass eine **zweite** Aktion nur laufen kann, wenn die **erste** Aktion fertig ist.

Beschrieben werden Abläufe in **hPDL**, eine XML basiert Process Definition Language (PDL).



# Oozie Abläufe

Ein Ablauf kann sowohl **Kontroll-** als auch **Aktionsknoten** enthalten. Erstere definieren das **Anfang** und **Ende** des Ablaufs und ermöglichen den Ablauf selbst zu **steuern** (Entscheidung, Aufteilung und Zusammenführung).

Weiterhin kann ein Ablauf auch **parametrisiert** werden, was es erlaubt **denselben** Ablauf **mehrfach** zu verwenden, in dem man z. B. die Ein- und Ausgabe Verzeichnisse über Parameter **anpasst**.





# Oozie Aktionen

Die **Aktionen** eines Ablaufs werden immer **außerhalb** von Oozie ausgeführt, normalerweise auf einem **dedizierten** Cluster. Wenn eine Aktion **fertig** ist **benachrichtigt** sie Oozie, welches dann die **nächste** Aktion starten kann.

Sie sind somit die **kleinste** Einheit an Arbeit welche Oozie unterstützt und stoßen die eigentliche **Verarbeitung** an. Oozie **liefert** Aktionen bereits **mit**, sie können aber **beliebig** durch Eigene **ergänzt** werden.



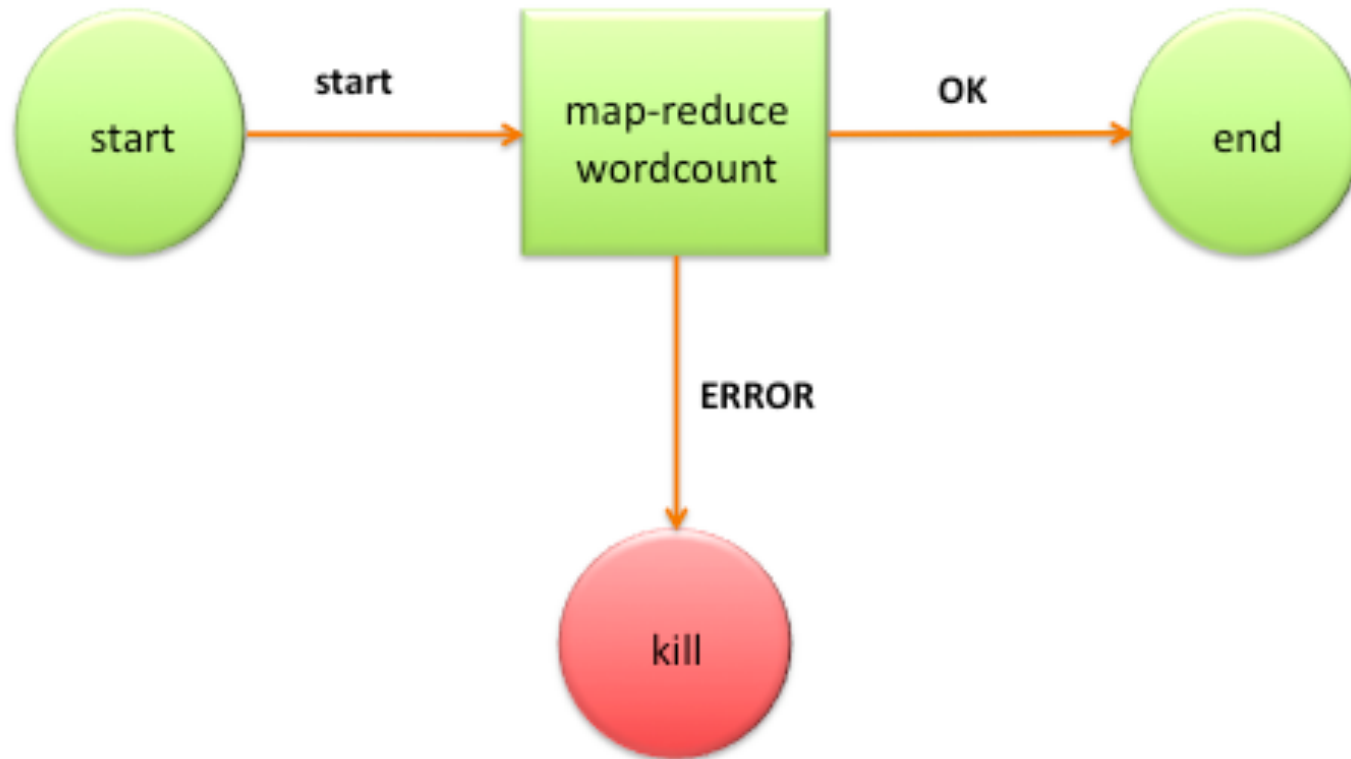
# Oozie Aktionen

Die **mitgelieferten** Aktionen sind unter anderem

- MapReduce
- Pig und Hive
- HDFS Operationen
- SSH
- HTTP
- E-Mail
- Oozie Unterabläufe



# Beispiel: Oozie Ablauf



Hinweis: Rund bedeutet Kontrollknoten, eckig sind die Aktionsknoten



# Beispiel: Oozie Ablauf

## Hinweise:

Es gibt immer nur **einen** einzigen **Start** und **Ende** Knoten, d. h. der Graph **muss** so definiert werden, das dies **gegeben** ist.

Alle **benutzten** Parameter müssen **gesetzt** sein über z. B. eine entsprechende Properties Datei, ansonsten **läuft** der Ablauf erst gar **nicht** an.



# Beispiel: Oozie Ablauf

```
<workflow-app name='wordcount-wf'
xmlns="uri:oozie:workflow:0.1">
  <start to='wordcount' />
  <action name='wordcount'>
    <map-reduce>
      <job-tracker>${jobTracker}</job-tracker>
      <name-node>${nameNode}</name-node>
      <configuration>
        <property>
          <name>mapred.mapper.class</name>
          <value>org.myorg.WordCount.Map</value>
        </property>
      </configuration>
    </map-reduce>
  </action>
</workflow-app>
```



# Beispiel: Oozie Ablauf

```
<property>
  <name>mapred.reducer.class</name>
  <value>org.myorg.WordCount.Reduce</value>
</property>
<property>
  <name>mapred.input.dir</name>
  <value>${inputDir}</value>
</property>
<property>
  <name>mapred.output.dir</name>
  <value>${outputDir}</value>
</property>
</configuration>
</map-reduce>
```

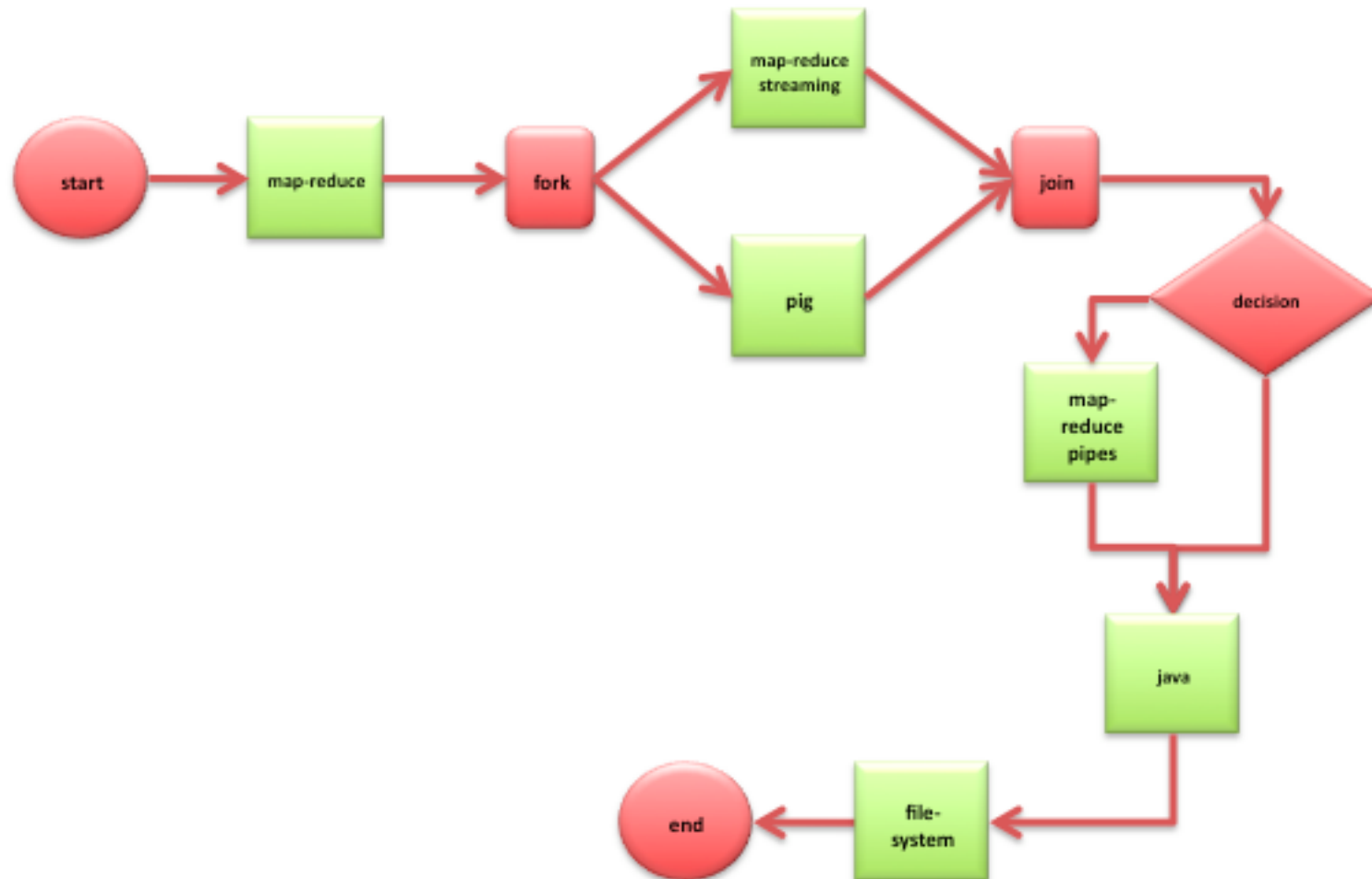


# Beispiel: Oozie Ablauf

```
<ok to='end' />
<error to='end' />
</action>
<kill name='kill'>
  <message>Something went wrong: \
    ${wf:errorCode('wordcount')}</message>
</kill/>
<end name='end' />
</workflow-app>
```



# Beispiel: Oozie Ablauf #2







# Ablauf Server

Oozie enthält einen **REST** basierten Server (läuft auf Apache Tomcat) mit einer **einfachen** webbasierten Oberfläche. Die Oberfläche kann nur **lesend** zugreifen, denn alle Operationen **müssen** über die Kommandozeile (**CLI**) abgeschickt werden. Die CLI ruft wiederum die **Funktionen** der REST API auf.

Der Server **führt** Abläufe aus und **überwacht** deren Status. Die Informationen legt Oozie's Server in einer relationalen Datenbank ab.

Eine **Alternative** stellt HUE dar (in VM enthalten).



# Ablauf Anwendung

Es werden mindestens **drei** Komponenten für einen **Ablauf** benötigt:

1. **Datei `workflow.xml`**

Beinhaltet die Ablauf **Definition** in hPDL

2. **Bibliotheken**

**Optionales** Verzeichnis „`lib/`“ für JAR und SO Dateien

3. **Properties Datei**

Enthält die Parameter für den Ablauf (für die `workflow.xml`) und **muss** mindestens `oozie.wf.application.path` **setzen**



# Ablauf Anwendung

## Beispiel für eine Properties Datei:

```
nameNode=hdfs://localhost:8020
```

```
jobTracker=localhost:8021
```

```
queueName=default
```

```
inputDir=${nameNode}/data.in
```

```
outputDir=${nameNode}/out
```

```
user.name=training
```

```
oozie.wf.application.path=\  
${nameNode}/user/${user.name}/oozie/  
workflow/wordcount/
```



# Ablauf Anwendung

Die **genannten** Komponenten (Dateien) werden in ein Verzeichnis (gesetzt mit `oozie.wf.application.path`) in HDFS **kopiert** und dann über die Kommandozeile (der API) **eingereicht**.

Danach kann der Ablauf über die Weboberfläche oder Kommandozeile (oder eigenem Code) **überwacht** werden.

Es gibt noch **weitere** Optionen, welche z. B. es erlauben von Oozie **aktiv** benachrichtigt zu werden, wenn ein Ablauf **beendet** wird.



# Ablauf Abschieken

Zum Abschieken eines Ablaufs:

```
$ oozie job -run -config job.properties \  
    -oozie http://localhost:11000/oozie/  
Workflow ID: 00123-123456-oozie-wrkf-W
```

Zum Prüfen des Status eines Ablaufs:

```
$ oozie job -info 00123-123456-oozie-wrkf-W \  
    -oozie http://localhost:11000/oozie/
```

-----  
Workflow Name: test-wf

App Path: hdfs://localhost:11000/user/your\_id/oozie/  
Workflow job status [RUNNING]

...  
-----



# Ablauf Fortsetzen

Wenn Aktionen in einem Ablauf **fehlschlagen**, dann können diese **wiederholt** werden. Zuerst aber **beendet** sich der ganze Ablauf **gemäß** der DAG basierten **Definition**. Dann kann ein Ablauf entweder mit

`oozie.wf.rerun.failnodes=true` oder  
`oozie.wf.rerun.skip.nodes=<k1>, ...`  
und dem Parameter `-rerun` wieder gestartet werden. Diese beiden Optionen sind einander **ausschließend**.



# Oozie Koordinatoren

Ein **weiteres** Konzept in Oozie sind die Koordinatoren (**Coordinators**). Diese erlauben es Abläufe mit **Zeit-** oder **Dateiauslösern** zu starten. Dazu wird eine **weitere** XML Datei in dem Ablauf Verzeichnis **erstellt**, welches diese Bedingungen **definiert**.

Dies ist sinnvoll Abläufe **automatisiert** und **wiederholt** laufen zu lassen. Dadurch kann man beispielsweise auch Abläufe **verknüpfen**.



# Oozie Koordinatoren

Koordinatoren sind **spezielle** Abläufe, werden aber wie die gerade gezeigten, **reinen** Abläufe genauso über die CLI abgeschickt.

Im Prinzip ist ein Koordinator ein **dauerhaft** laufender Prozess im Oozie Server, welcher je nach Definition den **enthalten** Ablauf anstößt.

Es gelten also die **gleichen** Kommandos für die Handhabung der Koordinatoren, wie bei den **reinen** Abläufen.





# Zeitbasierte Koordinatoren

Wenn ein Ablauf zu bestimmten **Zeiten** laufen soll, kann man eine **zeitbasierte** Koordinator Definition erstellen.

Es gibt aber nur das Konzept einer **Frequenz**, **nicht** das der **zeitgenauen** Ausführung (also nicht wie z. B. **CRON**).

Alle Koordinator Definitionen **müssen** in einer Datei namens `coordinator.xml` im **Hauptverzeichnis** der Anwendung gespeichert werden.



# Zeitbasierte Koordinatoren

Beispiel:

```
<coordinator-app name="coordinator1"
frequency="{frequency}" start="{start"
{startTime}" end="{endTime}"
timezone="{timezone}"
xmlns="uri:oozie:coordinator:0.1">
  <action>
    <workflow>
      <app-path>{workflowPath}</app-path>
    </workflow>
  </action>
</coordinator-app>
```



# Zeitbasierte Koordinatoren

Genauso wie auch die eigentliche `workflow.xml` kann auch die `coordinator.xml` Datei parametrisiert werden. Zum Beispiel:

...

`frequency=60`  In Minuten

`startTime=2012-08-31T20\:20Z`

`endTime=2013-08-31T20\:20Z`

`timezone=GMT+0530`

`workflowPath=${nameNode}/user/${  
user.name}/oozie/workflow/wordcount/`



# Dateibasierte Koordinatoren

Wenn man in die `coordinator.xml` um **dateibasierte** Elemente erweitert, kann definiert werden, dass Oozie einen Ablauf nach **Eintreffen** von **bestimmten** Eingabedateien **automatisch** ausführt.

So können die **Ausgabedateien** eines Ablaufs als **Eingabe** eines weiteren dienen. Oder Daten, welche über **Flume** gespeichert werden, können nachdem ein Datei voll ist verarbeitet werden.



# Dateibasierte Koordinatoren

## Beispiel Definition:


```
...
<datasets>
  <dataset name="input1" frequency="${datasetfrequency}" \
    initial-instance="${datasetinitialinstance}" \
    timezone="${datasettimezone}">
    <uri-template>${dataseturitemplate}/${YEAR}/${MONTH}/ \
      ${DAY}/${HOUR}/${MINUTE}</uri-template>
    <done-flag> </done-flag>
  </dataset>
</datasets>
<input-events>
  <data-in name="coordInput1" dataset="input1">
    <start-instance>${inputeventstartinstance}</start-instance>
    <end-instance>${inputeventendinstance}</end-instance>
  </data-in>
</input-events>
...
```



# Dateibasierte Koordinatoren

## Beispiel Properties:

...

datasetfrequency=15  In Minuten

datasetinitialinstance=2012-08-21T15:30Z

datasettimezone=UTC

dataseturitemplate=\${namenode}/srvs/s0001/in

inputeventstartinstance=\${coord:current(0)}

inputeventendinstance=\${coord:current(0)}

workflowPath=\${nameNode}/user/\${user.name}/oozie/workflow/  
wordcount/

...

inputDir= \${coord:dataIn('coordInput1')}

outputDir=\${nameNode}/out

oozie.coord.application.path=\${nameNode}/user/\${user.name}/  
coordOozie/coordinatorFileBased-events>



# Koordinator Abschicken

**Zum Abschicken eines Koordinator Ablaufs:**

```
$ oozie job -run -config job.properties \  
    -oozie http://localhost:11000/oozie/  
job: 00123-123456-oozie-hado-C
```

**Zum Anhalten eines Koordinator Ablaufs:**

```
$ oozie job -suspend 00123-123456-oozie-hado-C \  
    -oozie http://localhost:11000/oozie/
```

**Zum Starten eines angehaltenen Koordinator Ablaufs:**

```
$ oozie job -resume 00123-123456-oozie-hado-C \  
    -oozie http://localhost:11000/oozie/
```

**Zum Beenden eines Koordinator Ablaufs:**

```
$ oozie job -kill 00123-123456-oozie-hado-C \  
    -oozie http://localhost:11000/oozie/
```



# Oozie Bündel

Das **neueste** Konzept in Oozie sind die Bündel (**Bundles**), welche es erlauben **mehrere** Koordinatoren Abläufe als **Datenpipelines** zusammenzufassen. Dabei werden **Abhängigkeiten** zwischen den Abläufen definiert und diese dann von Oozie **eingehalten**.





# Mikro Pipelines

Im Gegensatz zu den Makro Pipelines, sind die **Mikro Pipelines** eher mit der eigentlichen **Datenbearbeitung** beschäftigt. Sie können aber sehr wohl auch **mehrere** Arbeitsabläufe (Jobs) anstoßen. Der Fokus liegt aber auf dem **Datenfluss** und der **Transformation** der Tupel. Eingabe und Ausgabe sind deshalb eben jene **Datenpaare** (oder Tupel) und die Schritte deren Verarbeitung.



# Morphlines

Ein Beispiel für eine Datenpipeline ist das **Morphline** Projekt. Es ermöglicht dem Anwender **kompakte, wiederholbare** Befehle auf Daten anzuwenden.

Im Prinzip ähneln Morphlines den Unix Pipelines (z. B. `cat text.txt | sort | uniq | wc -l`) sind aber effektiver, denn sie werden in einem Prozess ausgeführt.



# Morphlines

Das Projekt liefert schon **viele** Befehle mit, welche es erlauben **bekannte** Dateiformate **direkt** zu lesen, z. B. Log Dateien, Text, Avro, Sequence Dateien, JSON, HTML und XML.

Das Verarbeiten mit Morphlines kann in eigene Anwendungen **eingebettet** und beliebig **erweitert** werden.

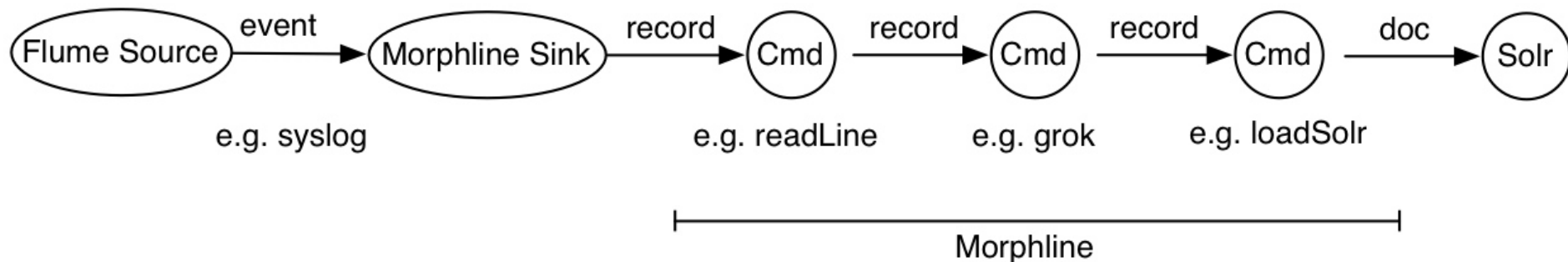
Definiert werden Morphlines im **HOCON** Format (Human-Optimized Config Object Notation).



# Prozessmodell

Das Prozessmodell erlaubt es Datensätze entgegen zu nehmen, diese zu zerlegen und über eine Kette an Befehlen zu verarbeiten.

Beispiel:





# Datenmodell

Das Datenmodell ist ähnlich dem MapReduce Gedanken: ein Datensatz wird in keinen, einen oder mehrere Datensätze zerlegt.

Im Unterschied zu MR ist ein Datensatz aber eine Menge an Feldern mit einem oder mehreren Werten. Also in Java ist dies:

```
Hashtable -> String -> Object[]
```

Dabei ist es egal welche Felder in welchen Datensätzen existiert (schemalos).



# Binäre Lasten

Es gibt auch die Möglichkeit binäre Daten in einem Datensatz mitzuführen. Dazu existiert der feste Feldname „\_attachment\_body“ welcher eine Java InputStream Instanz enthält (wenn gegeben).

Passend gibt es noch „\_attachment\_mimetype“ und „\_attachment\_charset“ welche den MIME Typ und Zeichensatznamen halten können. Dies ist der Behandlung von Anhängen in E-Mails nachempfunden.



# Benutzung

Morphline liefert schon einige Anwendungen mit, z. B. einen Flume Morphline Sink oder das MapReduce Indexer Tool.

Diese führen die Morphlines innerhalb des Prozesses des Gastgebersystems aus.



# Beispiel #1

## Beispiel „syslog“ Einträge:

```
<164>Feb  4 10:46:14 syslog sshd[607]: listening  
on 0.0.0.0 port 22.
```

## Zu erzeugender Ausgabedatensatz:

```
priority : 164  
timestamp : Feb  4 10:46:14  
hostname  : syslog  
program   : sshd  
pid       : 607  
msg       : listening on 0.0.0.0 port 22.  
message   : <164>Feb  4 10:46:14 syslog sshd[607]:  
listening on 0.0.0.0 port 22.
```





# Beispiel #1

```
morphlines : [  
  {  
    id : morphline1  
    importCommands : ["org.kitesdk.**", "org.apache.solr.**"]  
  
    commands : [  
      {  
        readLine {  
          charset : UTF-8  
        }  
      }  
    ]  
  }  
  ...  
]
```



# Beispiel #1

```
{
  grok {
    dictionaryFiles : [src/test/resources/grok-dictionaries]
    expressions : {
      message : "\"\"<{%{POSINT:priority}>{%{SYSLOGTIMESTAMP:timestamp} }%
{SYSLOGHOST:hostname} {%{DATA:program} (?:\[{%{POSINT:pid}\})?: %
{GREEDYDATA:msg}\"\"\"
    }
  }
}
{
  convertTimestamp {
    field : timestamp
    inputFormats : ["yyyy-MM-dd'T'HH:mm:ss'Z'", "\"MMM d HH:mm:ss\""]
    inputTimezone : America/Los_Angeles
    outputFormat : "yyyy-MM-dd'T'HH:mm:ss.SSS'Z'"
    outputTimezone : UTC
  }
}
```



# Beispiel #1

```
{
  sanitizeUnknownSolrFields {
    # Location from which to fetch Solr schema
    solrLocator : {
      collection : collection1      # Name of solr collection
      zkHost : "127.0.0.1:2181/solr" # ZooKeeper ensemble
    }
  }
}
{ logInfo { format : "output record: {}", args : ["@{}"] } }
{
  loadSolr {
    solrLocator : {
      collection : collection1      # Name of solr collection
      zkHost : "127.0.0.1:2181/solr" # ZooKeeper ensemble
    }
  }
}
}
]
}
```



# Beispiel #2

## Beispiel „log4j“ Einträge:

```
juil. 25, 2012 10:49:40 AM hudson.triggers.SafeTimerTask run ok
juil. 25, 2012 10:49:46 AM hudson.triggers.SafeTimerTask run failed
com.amazonaws.AmazonClientException: Unable to calculate a request signature
    at com.amazonaws.auth.AbstractAWSSigner.signAndBase64Encode(AbstractAWSSigner.java:71)
    at java.util.TimerThread.run(Timer.java:505)
Caused by: com.amazonaws.AmazonClientException: Unable to calculate a request signature
    at com.amazonaws.auth.AbstractAWSSigner.sign(AbstractAWSSigner.java:90)
    at com.amazonaws.auth.AbstractAWSSigner.signAndBase64Encode(AbstractAWSSigner.java:68)
    ... 14 more
Caused by: java.lang.IllegalArgumentException: Empty key
    at javax.crypto.spec.SecretKeySpec.<init>(SecretKeySpec.java:96)
    at com.amazonaws.auth.AbstractAWSSigner.sign(AbstractAWSSigner.java:87)
    ... 15 more
juil. 25, 2012 10:49:54 AM hudson.slaves.SlaveComputer tryReconnect
```



# Beispiel #2

## Beispiel „log4j“ Einträge:

```
juil. 25, 2012 10:49:40 AM hudson.triggers.SafeTimerTask run ok
juil. 25, 2012 10:49:46 AM hudson.triggers.SafeTimerTask run failed
com.amazonaws.AmazonClientException: Unable to calculate a request signature
    at com.amazonaws.auth.AbstractAWSSigner.signAndBase64Encode(AbstractAWSSigner.java:71)
    at java.util.TimerThread.run(Timer.java:505)
Caused by: com.amazonaws.AmazonClientException: Unable to calculate a request signature
    at com.amazonaws.auth.AbstractAWSSigner.sign(AbstractAWSSigner.java:90)
    at com.amazonaws.auth.AbstractAWSSigner.signAndBase64Encode(AbstractAWSSigner.java:68)
    ... 14 more
Caused by: java.lang.IllegalArgumentException: Empty key
    at javax.crypto.spec.SecretKeySpec.<init>(SecretKeySpec.java:96)
    at com.amazonaws.auth.AbstractAWSSigner.sign(AbstractAWSSigner.java:87)
    ... 15 more
juil. 25, 2012 10:49:54 AM hudson.slaves.SlaveComputer tryReconnect
```



# Morphlines

```

morphlines : [{
  id : morphline1
  importCommands : ["com.cloudera.**", "org.apache.solr.**"]
  commands : [{
    readMultiLine {
      regex : "(^.+Exception: .+)|(^\\s+at .+)|(^\\s+... \\d+
more)|(^\\s*Caused by:.)"
      what : previous
      charset : UTF-8
    }
  }
  { logDebug { format : "output record: {}", args :["@{}"] } }
  { loadSolr {}
  }
}]

```



# Eingebettete Ausführung

```
/** Usage: java ... <morphlines.conf> <dataFile1> ... <dataFileN> */
public static void main(String[] args) {
    // compile morphlines.conf file on the fly
    File configFile = new File(args[0]);
    MorphlineContext context = new MorphlineContext.Builder().build();
    Command morphline = new Compiler().compile(
        configFile, null, context, null);

    // process each input data file
    Notifications.notifyBeginTransaction(morphline);
    for (int i = 1; i < args.length; i++) {
        InputStream in = new FileInputStream(new File(args[i]));
        Record record = new Record();
        record.put(Fields.ATTACHMENT_BODY, in);
        morphline.process(record);
        in.close();
    }
    Notifications.notifyCommitTransaction(morphline);
}
```



# Informationsarchitektur

Nachdem betrachten der **einzelnen** Werkzeuge zur Datenverarbeitung und dem **Bilden** von Verarbeitungsketten (Pipelines), bleibt als **weiteres** Thema die **Organisation** von Daten.

Speziell im **Mandantenbetrieb** ist es wichtig Daten **geordnet** und **strukturiert** anzulegen, damit Fachabteilungen ihre und die **Daten** anderen Abteilungen **finden** und **nutzen** können. Dies ist die **Informationsarchitektur**.





# Informationsarchitektur

Für Lösungen basierend auf Hadoop ist dies mit Diensten (Services) abdeckbar. Dazu wird jeder Anwendungsfall in Benutzergruppen abgebildet, welche es für normaler Nutzer und Administratoren gibt. In HDFS z. B. werden dann die Dateien mit den entsprechenden Rechten in speziellen Verzeichnissen pro Dienst oder Anwendergruppe abgelegt.

Beispiele: `/srvs/s1/` oder `/system/etl/sales`



# Informationsarchitektur

**Unterhalb** der Dienst-/Gruppenebene werden dann Verzeichnisse für eingehende (**incoming**), ausgehende (**complete**), fehlerhafte (**failed**) und gerade bearbeitete Daten (**working**) abgebildet.

Im Arbeitsverzeichnis (working) wird **pro** Verarbeitungslauf ein Verzeichnis mit **Zeitstempel** (oder z. B. **Oozie Job ID**) angelegt. Damit können **mehrere** Abläufe **parallel** laufen **ohne** sich zu überschneiden.



# Informationsarchitektur

## Komplettes Beispiel:

/system

/system/etl/<group>/<process>

    /incoming

    /complete

    /failed

    /working

        /<epoch\_idx>

/incoming

/complete

/system/data/<dataset>



# Informationsarchitektur

**Dateien** in den Verzeichnissen werden dann den entsprechenden Gruppen **zugeordnet**, z. B. s1admin, s1user, s2admin, s2user,...

Wenn man **Ergebnisse** des Dienst #1 (s1) den Nutzern von Dienst #2 (s2) **zugänglich** machen möchte so müssen lediglich die passenden Benutzer der Gruppe s2 mit der **zusätzlichen** Gruppe s1user versehen werden.



# Informationsarchitektur

Die Benutzergruppen können **weiterhin** für die **Rechtevergabe** in den Verarbeitungsschlangen (**Queues**) verwendet werden. Damit kann einer Gruppe von Benutzern (oder einem automatisierten Dienst) genau **zugeordnet** werden, wie viele **Ressourcen** sie benutzen oder ob sie überhaupt Jobs laufen lassen dürfen.

**Fazit:** Hadoop selbst setzt **keine** Regeln, die Informationsarchitektur **muss** dies regeln.



# Einheit 6

An dieser Stelle endet die sechste Einheit, welche die Datenpipelines vorstellt und am Beispiel von Oozie und Morphlines vertieft.

In der nächsten Einheit befassen wir uns mit den abschließenden Fragen des Systemdesigns: Wie kombiniert man, zum Beispiel, Batch und Echtzeit Anwendungsfälle, oder wie bindet man die Ergebnisse in BI Lösungen ein?

Bis bald!



# Übung 6

## Ziele:

- Datenpipeline mit Oozie erstellen
- Datenpipeline mit Morphlines erstellen



# Übung 6

**Code:**

<https://github.com/larsgeorge/fh-muenster-bde-lesson-6>





# Quellen

- Oozie
  - Projekt: <http://oozie.apache.org/>
- Morphlines
  - Kite SDK:  
<http://kitesdk.org/docs/current/kite-morphlines/index.html>
  - Blog Post:  
[http://www.cloudera.com/content/cloudera-content/cloudera-docs/Search/latest/Cloudera-Search-User-Guide/csug\\_morphline\\_example.html](http://www.cloudera.com/content/cloudera-content/cloudera-docs/Search/latest/Cloudera-Search-User-Guide/csug_morphline_example.html)