

Big Data – ETL Workflow & Automation, Batch and Stream Processing

*Winter Semester 2019,
Cooperative State University Baden-Wuerttemberg*



Agenda – 04.11.2018

01

Presentation and Discussion: Exercise Of Last Lecture
Spark, PySpark and Jupyter

02

Batch and Stream Processing

A quick introduction to the principles, architectures and purposes of batch and stream processing in terms of big data.

03

Exercise – Working with PDI on Hadoop and Hive

Automate the process of downloading, importing, transforming and storing IMDb data within da Hadoop Cluster.

04

Exercise – Working with Apache Airflow on Hadoop and Hive

Automate the process of downloading, importing, transforming and storing IMDb data within da Hadoop Cluster.



Schedule

	<i>Lecture Topic</i>	<i>HandsOn</i>
14.10.2019 13:15-18:00 Ro. 1.18	About This Lecture, Introduction to Big Data, Setup Cloud Environment (Google Cloud)	HandsOn Hadoop, MapReduce and Hive(-QL)
21.10.2019 13:15-18:00 Ro. 1.18	(Non-)Functional Requirements Of Distributed Data-Systems, Data Models and Access	Partitioning with HDFS/Hive, HiveServer2
28.10.2019 13:15-18:00 Ro. 1.18	Challenges Of Distributed Data Systems: Replication and Partitioning	Spark, Scala and PySpark/Jupyter
04.11.2019 13:15-18:00 Ro. 1.18	ETL Workflow and Automation, Batch and Stream Processing	Pentaho Data Integration & Airflow
11.11.2019 13:15-18:00 Ro. 1.18	Practical Exam	Work On Practical Exam
18.11.2019 13:15-18:00 Ro. 1.18	Practical Exam Presentation	





Introduction to: **Batch and Stream Processing**

A quick introduction to the principles, architectures and purposes of batch and stream processing in terms of big data.



Batch vs Stream Processing

	Batch Processing	Stream Processing
Kind of data:	large, historic	volatile, live, stream
Runtime:	minutes, hours, days	real-time/near-real-time
Re-Execution:	possible	„impossible“



Batch Processing – Example Data Flow

Source Systems:

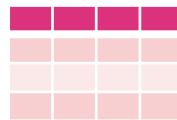
e.g. CRM, ERP, Webserver
Logfiles, ...



CSV Files



JSON Files

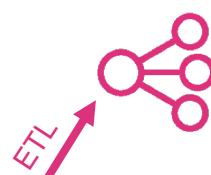


DB Tables

...

Data Lake:

e.g. Hadoop HDFS



ETL



ETL



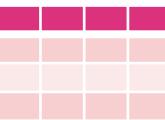
ETL

CSV Files



ETL

JSON Files



ETL

DB Tables

JSON Files

DB Tables

...

Data Warehouse:

e.g. PostgreSQL, Oracle,
MS SQL Server, DB2,
Informix...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

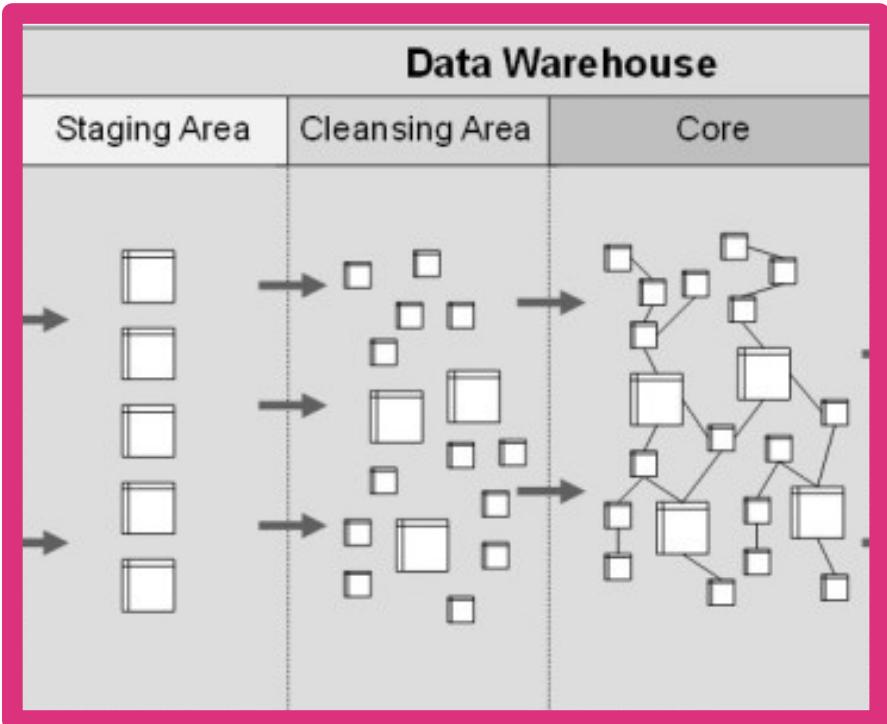
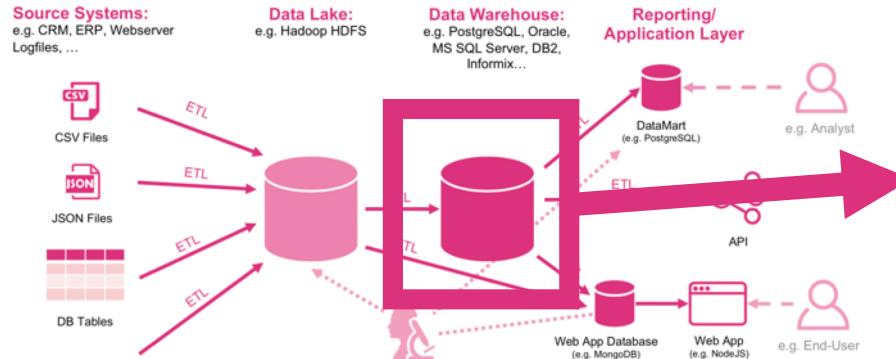
...

Batch Processing - ETL



ETL (Extract, Transform, Load) is a basic pattern for data processing, commonly known in data warehousing. It's all about extracting data from a source, transforming the data (e.g. by applying business rules or changing structures) and at the end writing/loading everything to a target (e.g. Hadoop HDFS, Hive, Relational Database, Data Warehouse, Data Mart etc.).

Batch Processing – Example Data Flow



Batch Processing – Dissociation Datawarehousing

- a Big Data data-system can be a **part** or **source** of a Data Warehouse, e.g.:
 - Data Lake or
 - Enterprise Data Hub
- A Data Warehouse is not Big Data

Data Warehouse

- handles mainly **structured data**
- suitable for **small amounts of data**
- focuses on **analytical and reporting purposes**
- 100% accuracy

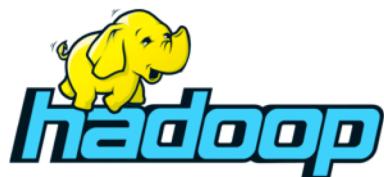
Big Data

- handles data in **any kind of structure**
- serves **broad variety** of data-driven **purposes** (e.g. analytical, data science, data-driven applications, ...)
- usually not about 100% accuracy



Distributed Batch Processing

Distributed Storage

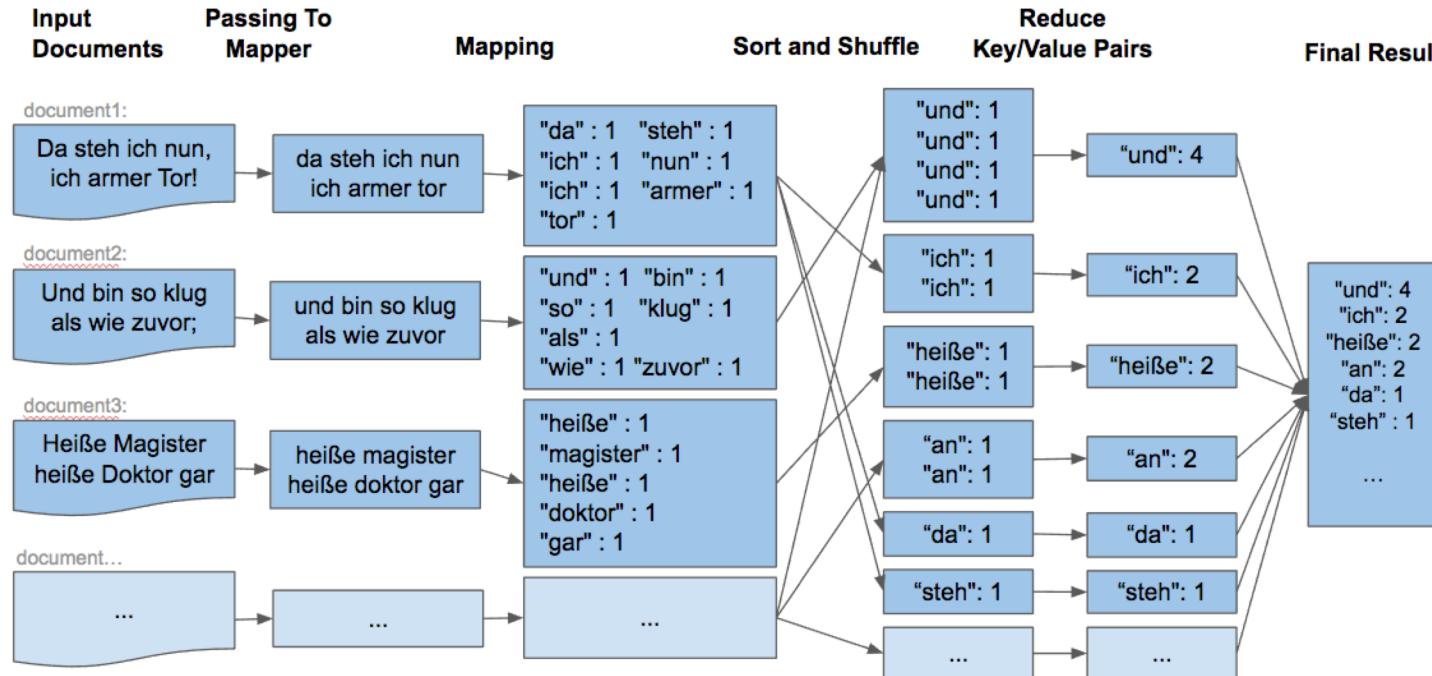


Distributed Processing



Batch Processing – MapReduce

- Programming paradigm for processing large datasets in parallel on a distributed cluster

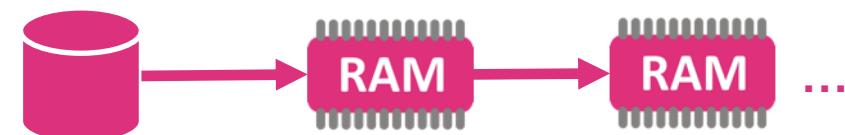


Batch Processing – MapReduce vs Spark

Hadoop MapReduce



Spark



- Good **reliability**
- Bad **performance**

- Bad **reliability**
- Good **performance**

Spark vs Flink



- **batch processing framework** that emulates stream processing
- **stream processing** = Execution of **micro batches**
- cuts event stream into **micro batches** and processes each batch
- **Latency:** (several) seconds
- **Maturity:** high, many libraries, huge community and developer base

- **stream processing framework** that emulates batch processing
- **batch processing** = bounded stream processing
- processes **each event** when it arrives
- **Latency:** milliseconds-seconds
- **Maturity:** medium, limited libraries, medium community and developer base

Spark vs Flink



- good choice for batch processing
 - stream processing:
 - if **reliability > latency**
- good choice for stream processing:
 - if **latency > reliability**

Stream Processing – Definition

Data Stream:

- is data made available over time in an incremental way
- created by:
 - static data (e.g. file or database read line-wise)
 - dynamic data (e.g. logs, sensors, function calls, ...)

Event:

- is an immutable record/item in a stream
- usually represented and encoded in e.g. JSON, XML, CSV or binary encoded
- pendants



Stream Processing – Use Cases/Data Sources

- User Interaction, e.g.:
 - webserver logfiles
 - tracking data like Google Analytics
 - recommendation systems (advertisement, personalization)
 - ...
- Sensor data
- Any API serving data in a stream, e.g.:
 - Twitter Postings API,
 - Message Queue/Broking Systems (e.g. Kafka, ZeroMQ, RabbitMQ...)
 - IoT
- Location Tracking (e.g. DriveNow, Car2Go, Google Maps...)
- ...

Message Broker/Queues

Why:

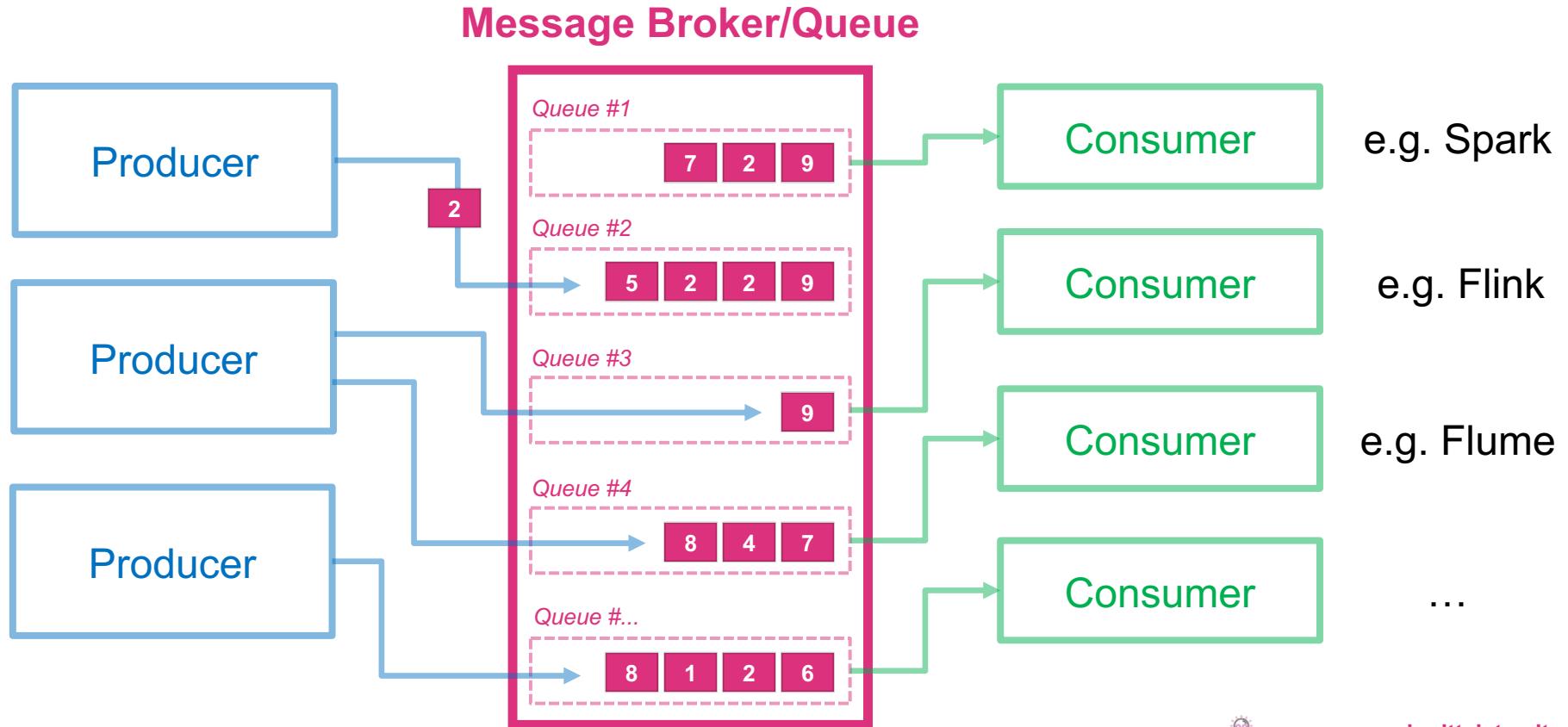
- decouples Producer and Receiver
- maintains queues for different topics
- temporarily persists messages
- Notifies subscribers on new messages
- Micro Services, IoT...

Examples:

- RabbitMQ,
- ZeroMQ,
- Kafka,
- ActiveMQ,
- Kestrel,
- ...



Message Broker/Queues



Message Broker/Queues vs Data-Systems

	Message Broker/Queue	Database
Persistence:	<ul style="list-style-type: none">- temporarily (delete once transmitted or after a certain timeframe)	<ul style="list-style-type: none">- persistent
Data Retrieval:	<ul style="list-style-type: none">- Subscription-based (even if e.g. Kafka supports query-based)	<ul style="list-style-type: none">- Query-based (execute query to receive data)
Communication:	<ul style="list-style-type: none">- Initiated by MB	<ul style="list-style-type: none">- Initiated by clients

Stream Processing – Windows

Sliding Window/
Gleitendes Fenster:



Stream:



Tumbling Window/
Rollierendes Fenster:



Hopping Window/
Springendes Fenster:



Session Window/
Sitzungs Fenster:



Stream Processing – Tumbling Windows

Data Stream:

- Fixed length
- Non-overlapping
- An event relates to exactly one window

Twitter Example:

- Calculate the count of tweets for every 4 seconds

```
SELECT count(*) as tweet_count FROM Twitter_Stream TIMESTAMP BY  
CreationTime GROUP BY TumblingWindow(second, 4)
```

Stream: ... 5 2 9 4 1 6 0 9 5 2 7 3 9 5 2 7 2 9 4 1 ...

Tumbling Window/
Rollierendes Fenster:



Stream Processing – Hopping Windows

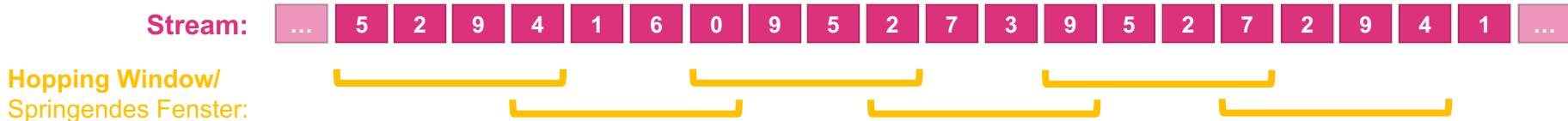
Data Stream:

- Fixed length
- Overlapping (with fix steps)
- An event relates to more than one window

Twitter Example:

- Every 3 seconds calculate the count of tweets for last 4 seconds

```
SELECT count(*) as tweet_count FROM Twitter_Stream TIMESTAMP BY  
CreationTime GROUP BY HoppingWindow(second, 4, 3)
```



Stream Processing – Sliding Windows

Data Stream:

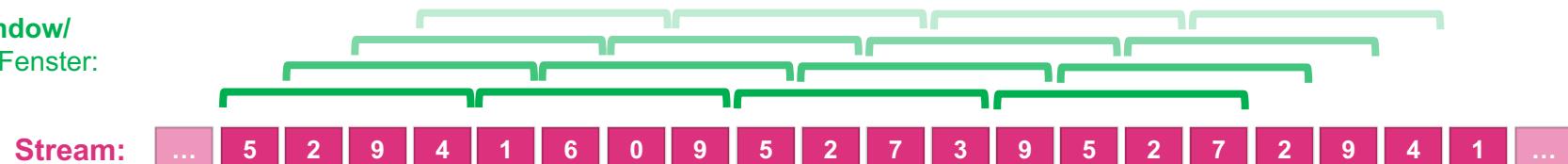
- Fixed length
- Overlapping (depending on event)
- An event can relate to more than one window
- Event-based, every window contains at least one event and is continuously slided forward

Twitter Example:

- Calculate count of tweets which have used a certain hashtag. Count number of tweets including hashtags, which were used more than 10 times within the last 4 seconds

```
SELECT HashTag, count(*) as tweet_count FROM Twitter_Stream TIMESTAMP BY  
CreationTime GROUP BY Hashtag, SlidingWindow(second, 4) HAVING count(*) > 10
```

Sliding Window/ Gleitendes Fenster:



Stream Processing – Session Windows

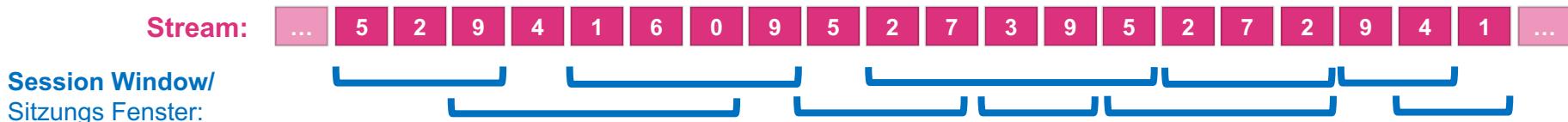
Data Stream:

- Arbitrary-length
- Overlapping possible
- An event can relate to more than one window
- Windows containing no events are filtered out
- Fix **start event** (e.g. login) and **end event** (logout or timeout)
- Max size usually limited

Twitter Example:

- Calculate count of tweets for certain hashtags that occur within less than 5 minutes to each other

```
SELECT HashTag, count(*) as tweet_count FROM Twitter_Stream TIMESTAMP BY  
CreationTime GROUP BY HashTag, SessionWindow(minute, 5)
```



Word Count on Stream – Apache Spark

```
// Initialize Spark Session
val spark = SparkSession
  .builder()
  .master("local")
  .appName("Socket_Streaming")
  .getOrCreate()

// Initialize Socket Stream
val socketStreamDf = spark.readStream
  .format("socket")
  .option("host", "localhost")
  .option("port", 4711)
  .load()

// parse, group, window and aggregate data
val words = socketStreamDf.as[String].flatMap(_.split(" "))
val wordCounts = words.groupBy("value").count()
val query = wordCounts.writeStream
  .outputMode("complete")
  .format("console")
  .start()

query.awaitTermination()
```

Input could also be **Kafka, Flume, Kinesis, Sockets, File, Custom Connectors...**

Output could also be **HDFS, Kafka, Kinesis, Console, ...**

Word Count on Stream – Apache Flink

```
// Initiate Execution Environment
val env = StreamExecutionEnvironment.getExecutionEnvironment

// Read data from socket on localhost, port: 4711
val text = env.socketTextStream("localhost", 4711, , '\n')

// parse, group, window and aggregate data
val word_counts = text
    // Split lines into 2-tuples: (word,1)
    .flatMap(_.toLowerCase.split("\\s"))
    .filter(_.nonEmpty)
    .map((_, 1))
    // group by word (tuple field "0")
    .keyBy(0)
    // sliding window: 5 second length, 1 second trigger interval
    .timeWindow(Time.seconds(5), Time.seconds(1))
    // sum up word count (tuple field "1")
    .sum(1)

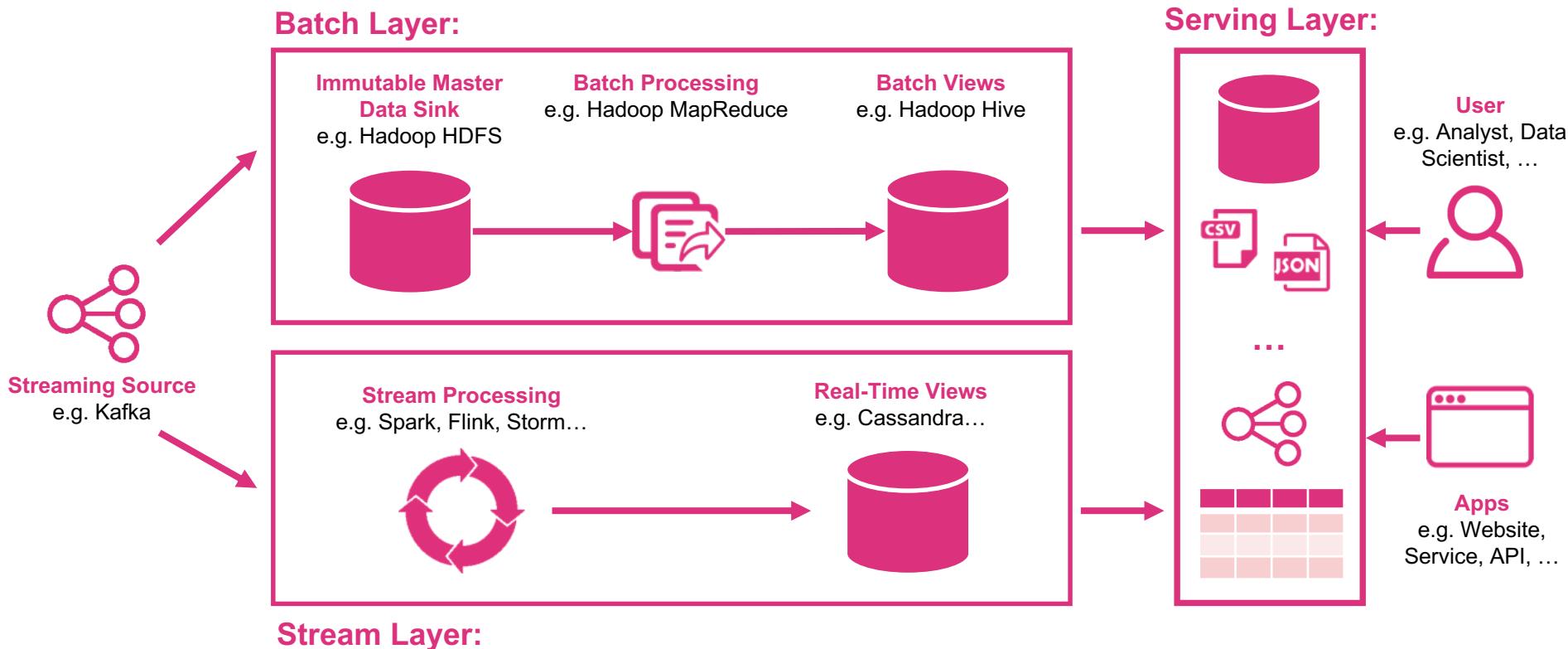
// print result (single thread)
word_counts.print().setParallelism(1)
}

// Execute
env.execute("Streaming Example: WordCount")
```

Input Could also be **Kafka, Kinesis, RabbitMQ, NiFi, ActiveMQ, Socket, File, Netty, Custom Connectors...**

Output could also be **HDFS, Kafka, Elasticsearch, Kinesis, RabbitMQ, NiFi, Akka, Redis, Cassandra, Flume, ActiveMQ, Custom Connectors... ...**

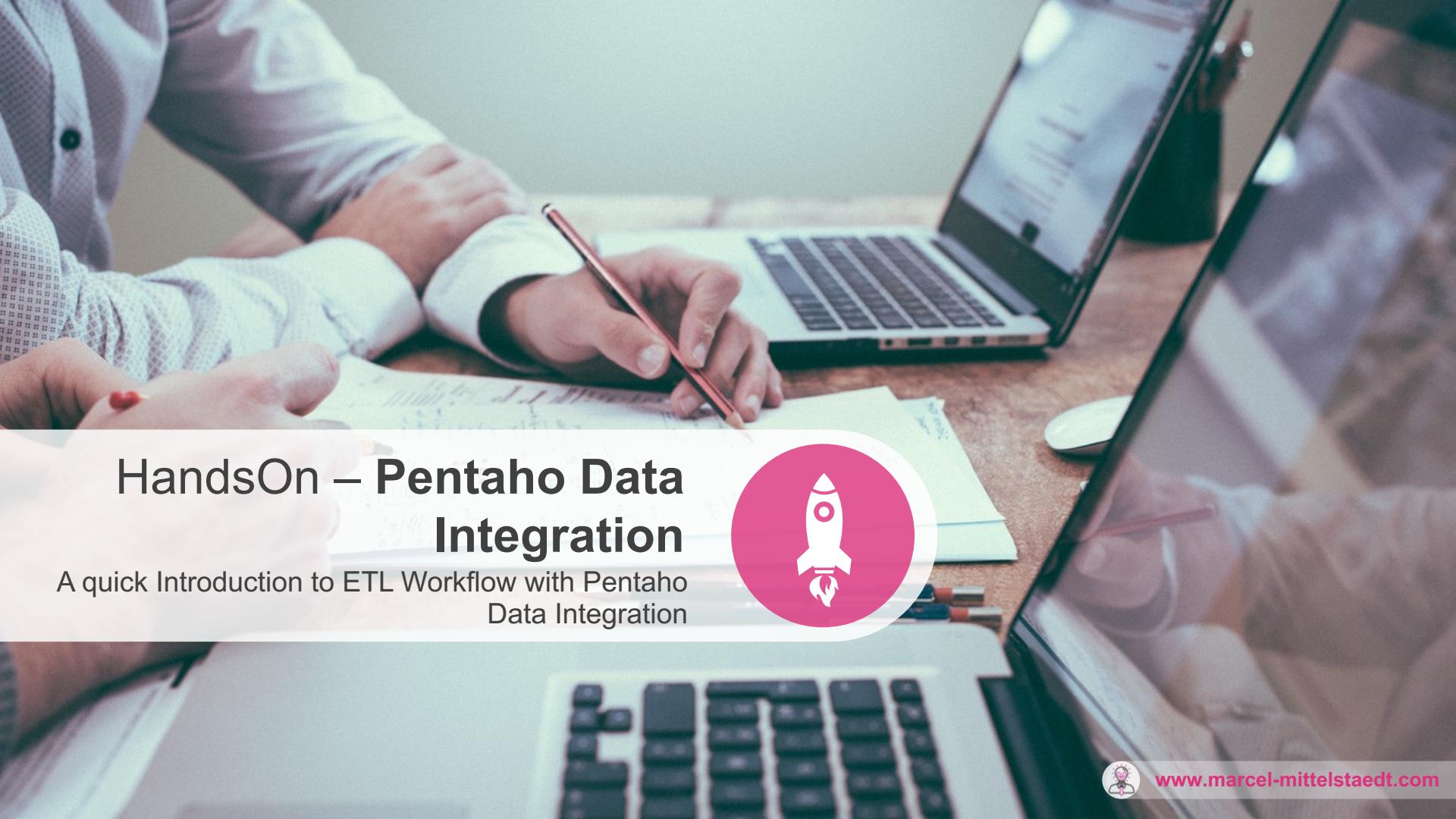
Batch&Stream Processing – Lambda Architecture



Break

TIME FOR
A
BREAK





HandsOn – Pentaho Data Integration

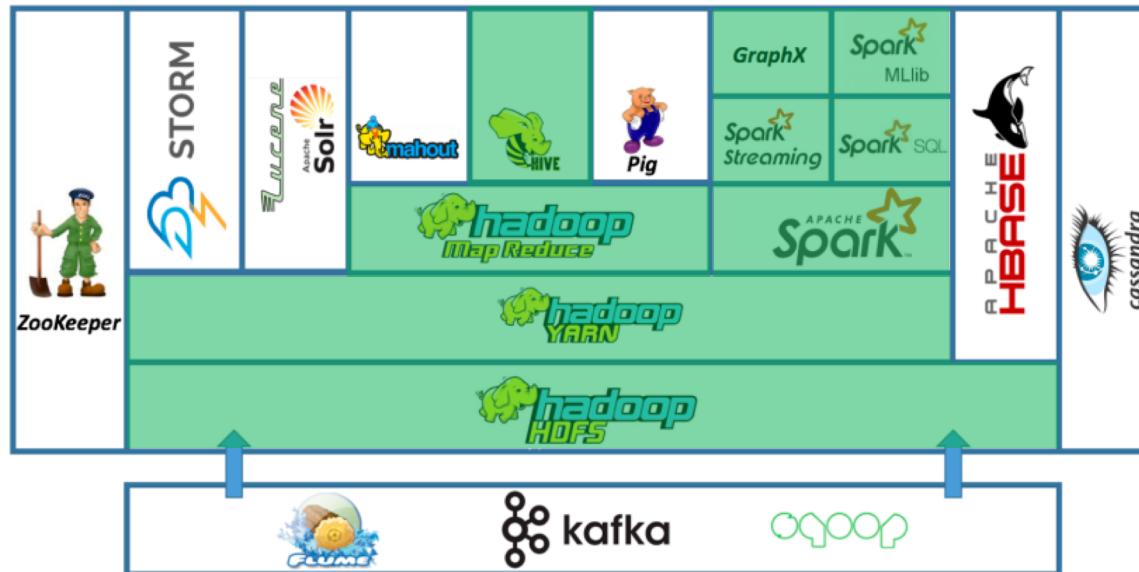
A quick Introduction to ETL Workflow with Pentaho Data Integration



The Hadoop Ecosystem



Today's
(exercise) focus



Exercises Preparation I

Install and Setup Pentaho Data Integration
First Steps



Download And Install PDI

1. Download Pentaho Data Integration (8.0)

```
https://sourceforge.net/projects/pentaho/
```

2. Extract:

```
unzip pdi-ce-8.1.0.0-365.zip
```



Start PDI (Spoon)

3. Start Pentaho Data Integration

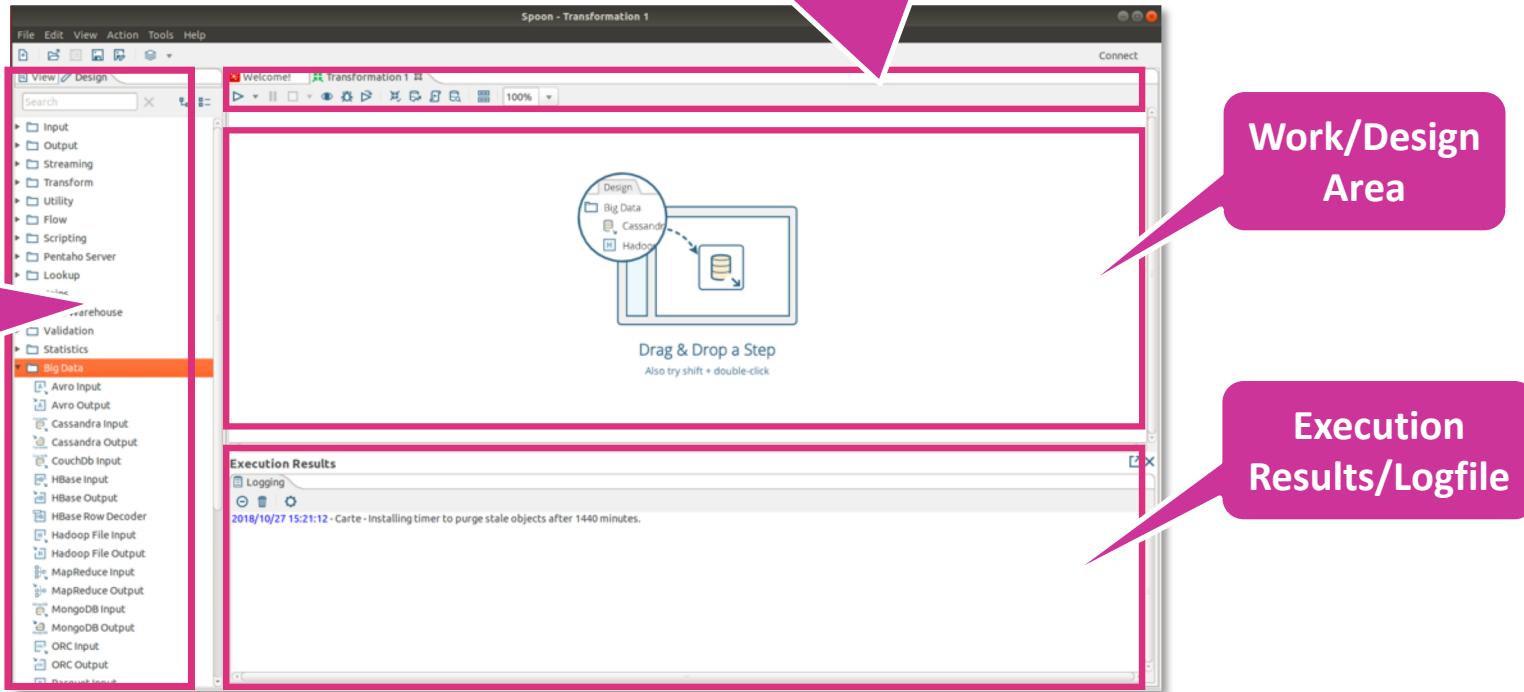
UNIX: /your/pentaho/directory/spoon.sh

Windows: /your/pentaho/directory/spoon.bat



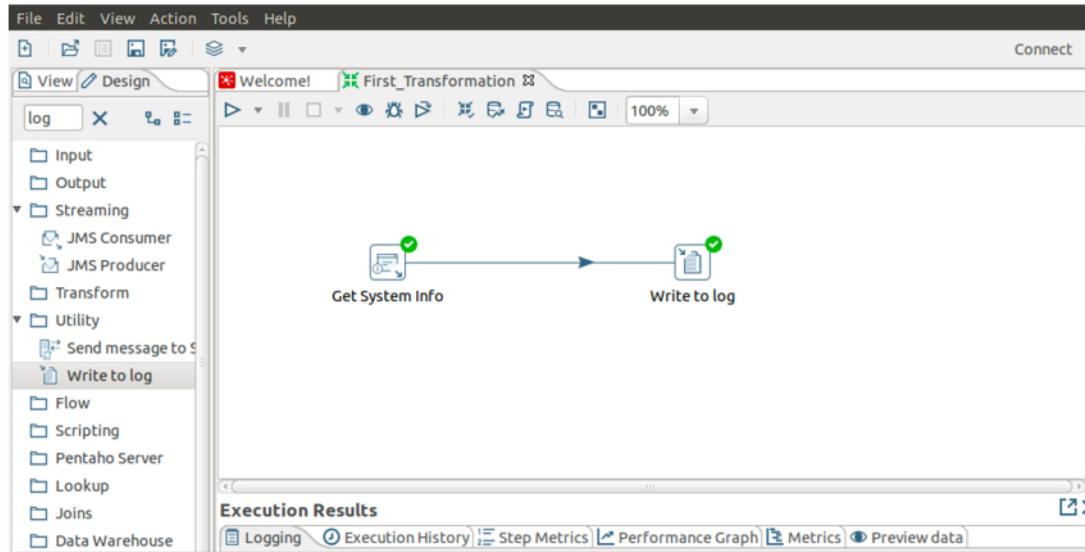
Spoon Interface

3. Start Pentaho Data Integration



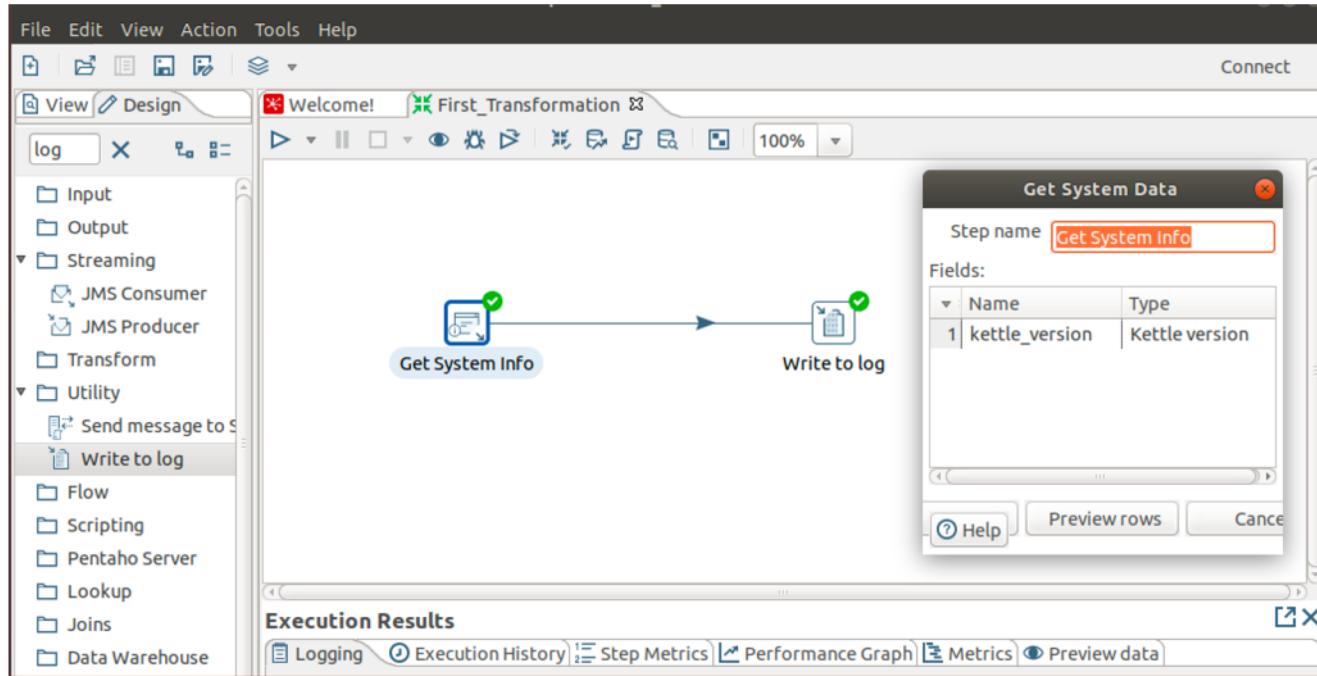
PDI Basics/Examples – First Transformation

1. Create new **Transformation** (Click: *File* → *New* → *Transformation*)
2. Drag&Drop step „Get System Info“ and „Write To Log“ into **Work/Design Area** and connect both steps:



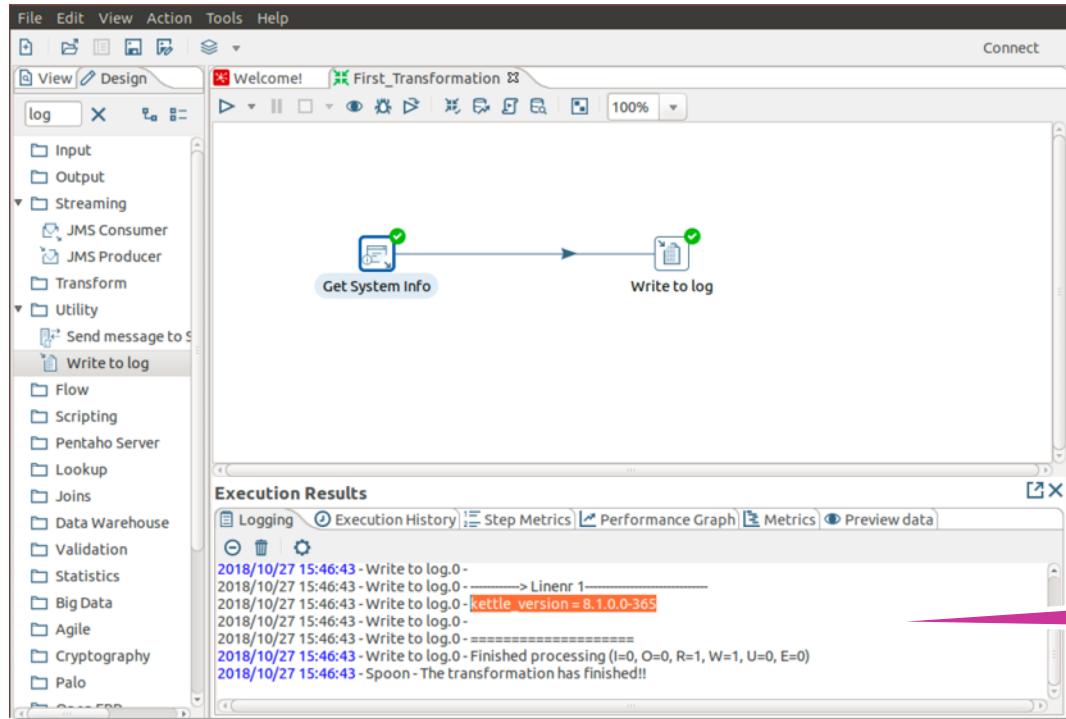
PDI Basics/Examples – First Transformation

3. Double Click „Get System Info“ step to configure step accordingly to kettle (PDI) version info:



PDI Basics/Examples – First Transformation

4. Save Transformation (*First_Transformation*). Press Run Button. See Results:



- results of „Get System Info“ step are redirected to „Write To Log“ step
- „Write to Log“ step will write results to execution log
- **transformations** are all about actual **data transformation** and data flow

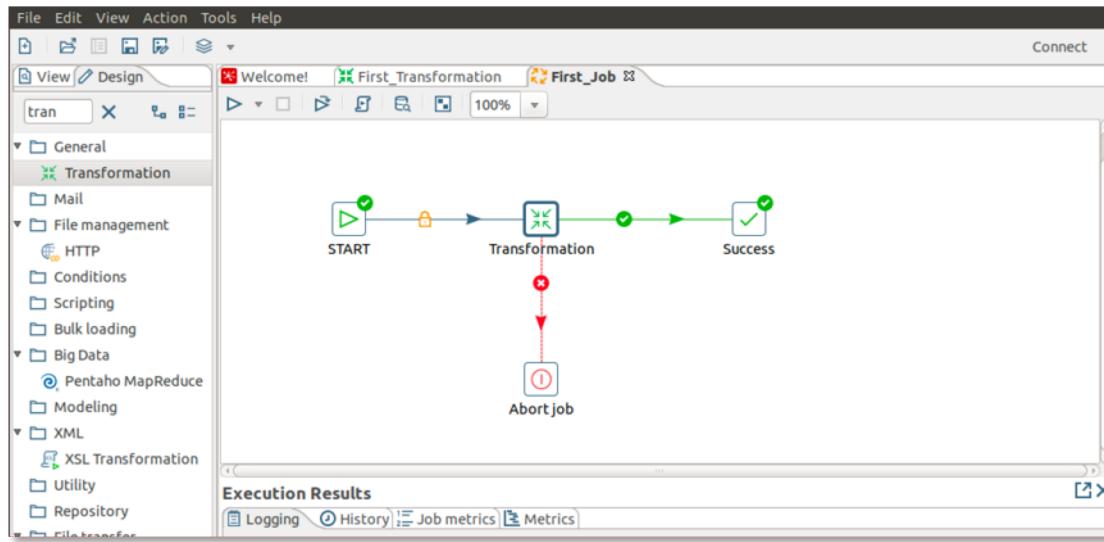
Results



PDI Basics/Examples – First Job

1. Create New **Job** (Click: *File* → *New* → *Job*)

2. Drag&Drop step „**START**“, „**Transformation**“, „**Abort Job**“ and „**Success**“ steps into **Work/Design Area** and connect all steps accordingly:



- Start of each Job



- End of a Job (if not successful)



- End of a Job (if successful)

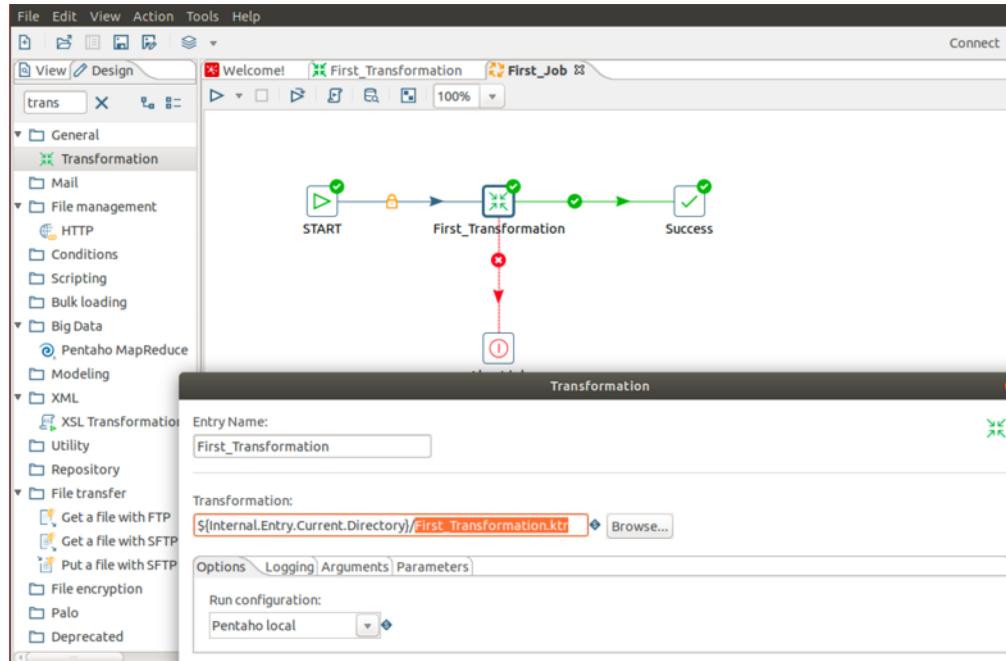


- Includes a Transformation



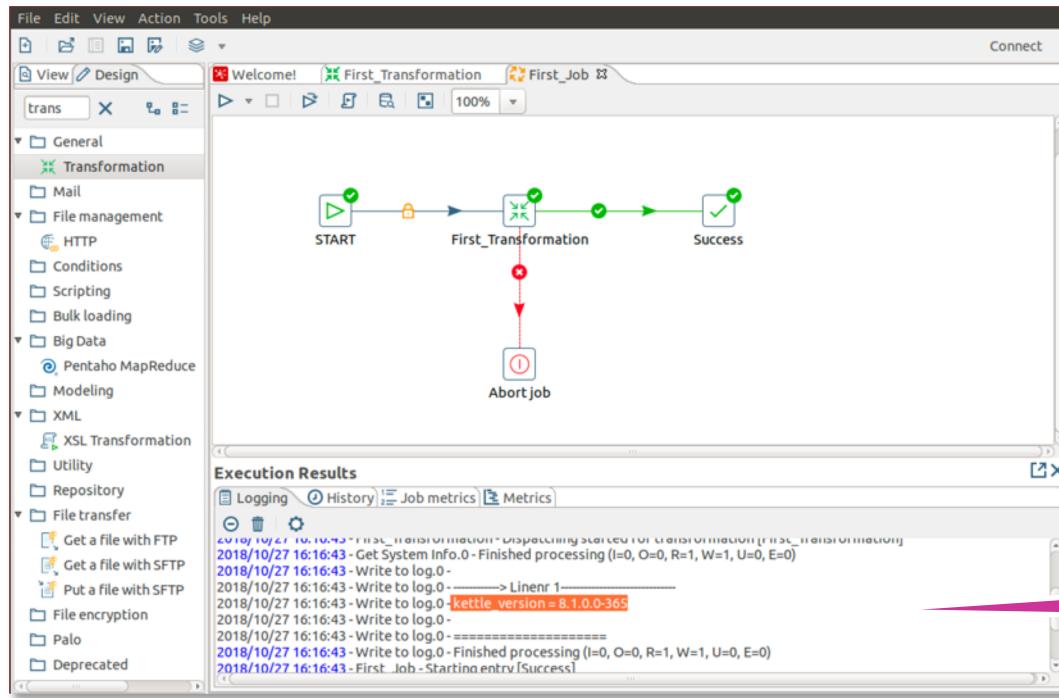
PDI Basics/Examples – First Job

3. Include previously created Transformation (First_Transformation) by double click on step „Transformation“ and including „First_Transformation.ktr“:



PDI Basics/Examples – First Job

4. Save Job (*First_Job*). Press Run Button. See Results:



- Job will execute previously created transformation (*First_Transformation.ktr*)
- Output of transformation will be appended to execution log of Job
- **jobs** are all about **workflows** of multiple **transformations** and **jobs**

Results

PDI Basics/Examples – Execute Jobs Using Kitchen

1. It's nice to run jobs locally during development, but how to run them on a remote server (e.g. in a productive manner)? This is easily achieved using kitchen.sh for Jobs:

```
/home/pentaho/pentaho/data-integration/pan.sh -file=/home/pentaho/pdi_jobs/First_Transformation.ktr
[...]
2019/11/03 19:47:59 - Kitchen - Finished!
2019/11/03 19:47:59 - Kitchen - Start=2019/11/03 19:47:34.003, Stop=2019/11/03 19:47:59.250
2019/11/03 19:47:59 - Kitchen - Processing ended after 25 seconds.
```

2. Or using pan.sh for Transformations:

```
/home/pentaho/pentaho/data-integration/kitchen.sh -file=/home/pentaho/pdi_jobs/First_Job.ktr
[...]
2019/11/03 19:45:47 - Pan - Start=2019/11/03 19:45:46.897, Stop=2019/11/03 19:45:47.023
2019/11/03 19:45:47 - Pan - Processing ended after 0 seconds.
2019/11/03 19:45:47 - First_Transformation -
2019/11/03 19:45:47 - First_Transformation - Step Get System Info.0 ended successfully, processed 1 lines. (- lines/s)
2019/11/03 19:45:47 - First_Transformation - Step Write to log.0 ended successfully, processed 1 lines. (- lines/s)
```



Break

TIME FOR
A
BREAK

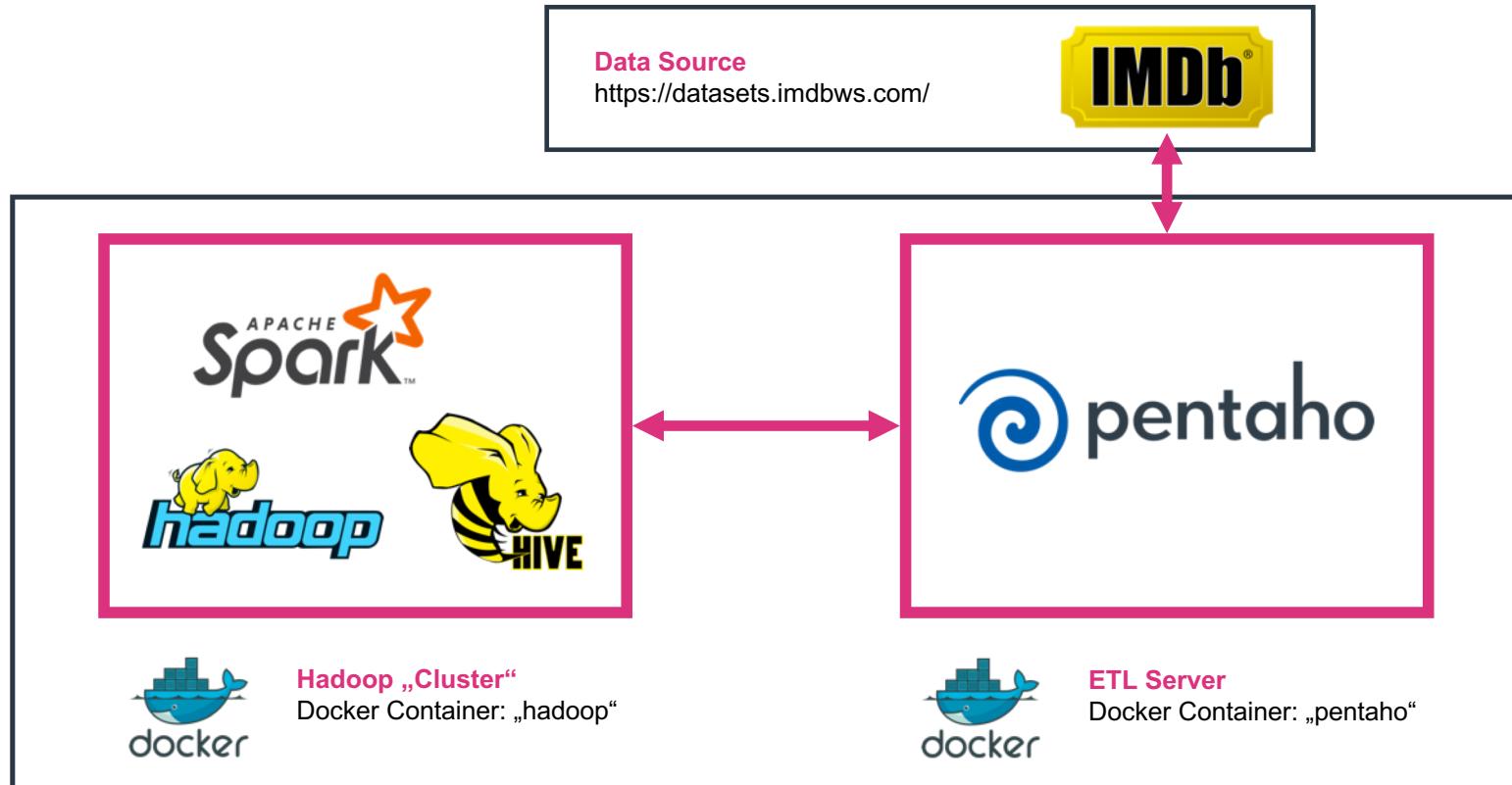


Exercises Preparation II

Start Hadoop/Hive/Spark Docker Image and a
Pentaho Data Integration Docker Image



What do we want to do?



Start Gcloud VM and Connect

1. Start Gcloud Instance:

```
gcloud compute instances start big-data
```

2. Connect to Gcloud instance via SSH (on Windows using Putty):

```
ssh hans.wurst@XXX.XXX.XXX.XXX
```



Start Hadoop/Hive/Spark Docker Container

1. Pull Docker Image:

```
docker pull marcelmittelstaedt/spark_base:latest
```

2. Start Docker Image:

```
docker run -dit --name hadoop \
-p 8088:8088 -p 9870:9870 -p 9864:9864 -p 10000:10000 \
-p 8032:8032 -p 8030:8030 -p 8031:8031 -p 9000:9000 \
-p 8888:8888 --net bigdatanet \
marcelmittelstaedt/spark_base:latest
```

3. Wait till first Container Initialization finished:

```
docker logs hadoop

[...]
Stopping nodemanagers
Stopping resourcemanager
Container Startup finished.
```



Start Hadoop/Hive/Spark Docker Container

4. Get into Docker container:

```
docker exec -it hadoop bash
```

5. Switch to hadoop user:

```
sudo su hadoop
```

```
cd
```

6. Start Hadoop Cluster:

```
start-all.sh
```

7. Start HiveServer2:

```
hiveserver2
```



Start Hadoop/Hive/Spark Docker Container

8. Start HiveServer2:

```
hive/bin/hiveserver2

2018-10-02 16:19:08: Starting HiveServer2
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/home/hadoop/hive/lib/log4j-slf4j-impl-2.10.0.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/home/hadoop/hadoop/share/hadoop/common/lib/slf4j-log4j12-1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.apache.logging.slf4j.Log4jLoggerFactory]
Hive Session ID = b8d1efb3-fc8c-4ec8-bdf0-6a9a41e2ddaa
Hive Session ID = 32503981-a5fd-497e-b887-faf3ec1e686e
Hive Session ID = 00f7eab4-5a29-4ce4-ad97-e90904d9206f
Hive Session ID = 100e54c5-14c6-4acc-b398-040152b08ebf
[...]
```



Start ETL (Pentaho) Docker Container

1. Pull Docker Image:

```
docker pull marcelmittelstaedt/pentaho:latest
```

2. Start Docker Image:

```
docker run -dit --name pentaho \
--net bigdatanet \
marcelmittelstaedt/pentaho:latest
```

3. Wait till first Container Initialization finished:

```
docker logs pentaho
[...]
Resolving deltas: 100% (692/692), done.
Checking out files: 100% (216/216), done.
Container Startup finished.
```



Start ETL (Pentaho) Docker Container

4. Get into Docker container:

```
docker exec -it pentaho bash
```

5. Switch to pentaho user:

```
sudo su pentaho
```

```
cd
```

Execute Transformations using pan.sh

1. Run first Transformation on ETL Server:

```
/home/pentaho/pentaho/data-integration/pan.sh -file=/home/pentaho/pdi_jobs/First_Transformation.ktr  
[...]  
2019/11/03 19:45:46 - Pan - Start of run.  
2019/11/03 19:45:46 - First_Transformation - Dispatching started for transformation [First Transformation]  
2019/11/03 19:45:47 - Get System Info.0 - Finished processing (I=0, O=0, R=1, W=1, U=0, E=0)  
2019/11/03 19:45:47 - Write to log.0 -  
2019/11/03 19:45:47 - Write to log.0 - -----> Linenr 1-----  
2019/11/03 19:45:47 - Write to log.0 - kettle_version = 8.0.0.0-28  
2019/11/03 19:45:47 - Write to log.0 -  
2019/11/03 19:45:47 - Write to log.0 - ======  
2019/11/03 19:45:47 - Write to log.0 - Finished processing (I=0, O=0, R=1, W=1, U=0, E=0)  
2019/11/03 19:45:47 - Pan - Finished!  
2019/11/03 19:45:47 - Pan - Start=2019/11/03 19:45:46.897, Stop=2019/11/03 19:45:47.023  
2019/11/03 19:45:47 - Pan - Processing ended after 0 seconds.  
2019/11/03 19:45:47 - First_Transformation -  
2019/11/03 19:45:47 - First_Transformation - Step Get System Info.0 ended successfully, processed 1 lines. (- lines/s)  
2019/11/03 19:45:47 - First_Transformation - Step Write to log.0 ended successfully, processed 1 lines. (- lines/s)
```



Execute Transformations using kitchen.sh

1. Run first Job on ETL Server:

```
/home/pentaho/pentaho/data-integration/kitchen.sh -file=/home/pentaho/pdi_jobs/First_Job.ktr  
[...]  
2019/11/03 19:47:59 - First_Job - Start of job execution  
2019/11/03 19:47:59 - First_Job - Starting entry [Transformation]  
2019/11/03 19:47:59 - Transformation - Loading transformation from XML file [file:///home/pentaho/pdi_jobs/First_Transformation.ktr]  
2019/11/03 19:47:59 - Transformation - Using run configuration [Pentaho local]  
2019/11/03 19:47:59 - Transformation - Using legacy execution engine  
2019/11/03 19:47:59 - First_Transformation - Dispatching started for transformation [First_Transformation]  
2019/11/03 19:47:59 - Get System Info.0 - Finished processing (I=0, O=0, R=1, W=1, U=0, E=0)  
2019/11/03 19:47:59 - Write to log.0 -  
2019/11/03 19:47:59 - Write to log.0 - -----> Linenr 1-----  
2019/11/03 19:47:59 - Write to log.0 - kettle_version = 8.0.0.0-28  
2019/11/03 19:47:59 - Write to log.0 -  
2019/11/03 19:47:59 - Write to log.0 - -----  
2019/11/03 19:47:59 - Write to log.0 - Finished processing (I=0, O=0, R=1, W=1, U=0, E=0)  
2019/11/03 19:47:59 - First_Job - Starting entry [Success]  
2019/11/03 19:47:59 - First_Job - Finished job entry [Success] (result=[true])  
2019/11/03 19:47:59 - First_Job - Finished job entry [Transformation] (result=[true])  
2019/11/03 19:47:59 - First_Job - Job execution finished  
2019/11/03 19:47:59 - Kitchen - Finished!  
2019/11/03 19:47:59 - Kitchen - Start=2019/11/03 19:47:34.003, Stop=2019/11/03 19:47:59.250  
2019/11/03 19:47:59 - Kitchen - Processing ended after 25 seconds.
```



Exercises Preparation III

A simple ETL examples of how to use an ETL Workflow tool (PDI) to integrate IMDb data



PDI IMDb Import

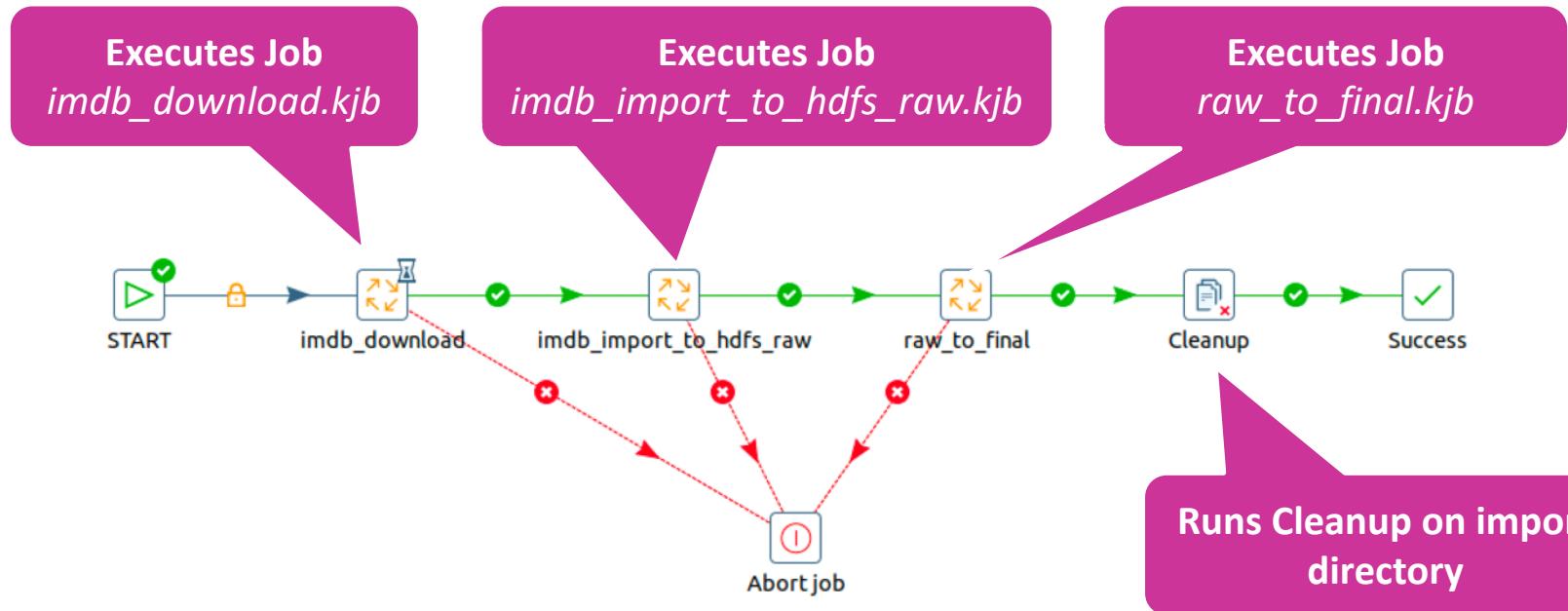
Now let's take a look at a real world example: **IMDB data**

https://github.com/marcelmittelstaedt/BigData/tree/master/exercises/winter_semester_2019-2020/06_pentaho

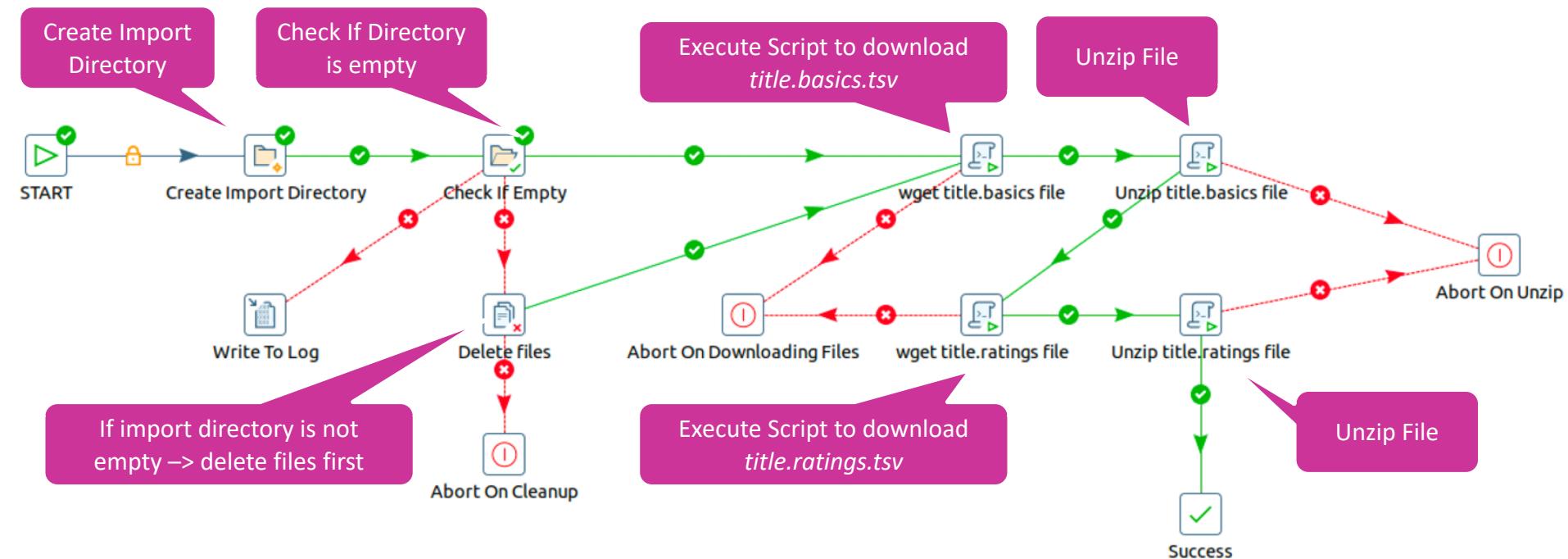
A very simple and naive ETL workflow which is able to run each day:

- **Download of IMDb data** to local filesystem
- Move Data to HDFS (raw directory/layer)
- Create Hive tables for **Raw Layer**
- Create and fill **Final Layer** (Hive tables) by raw layer tables, applying business rules and using dynamic partitioning

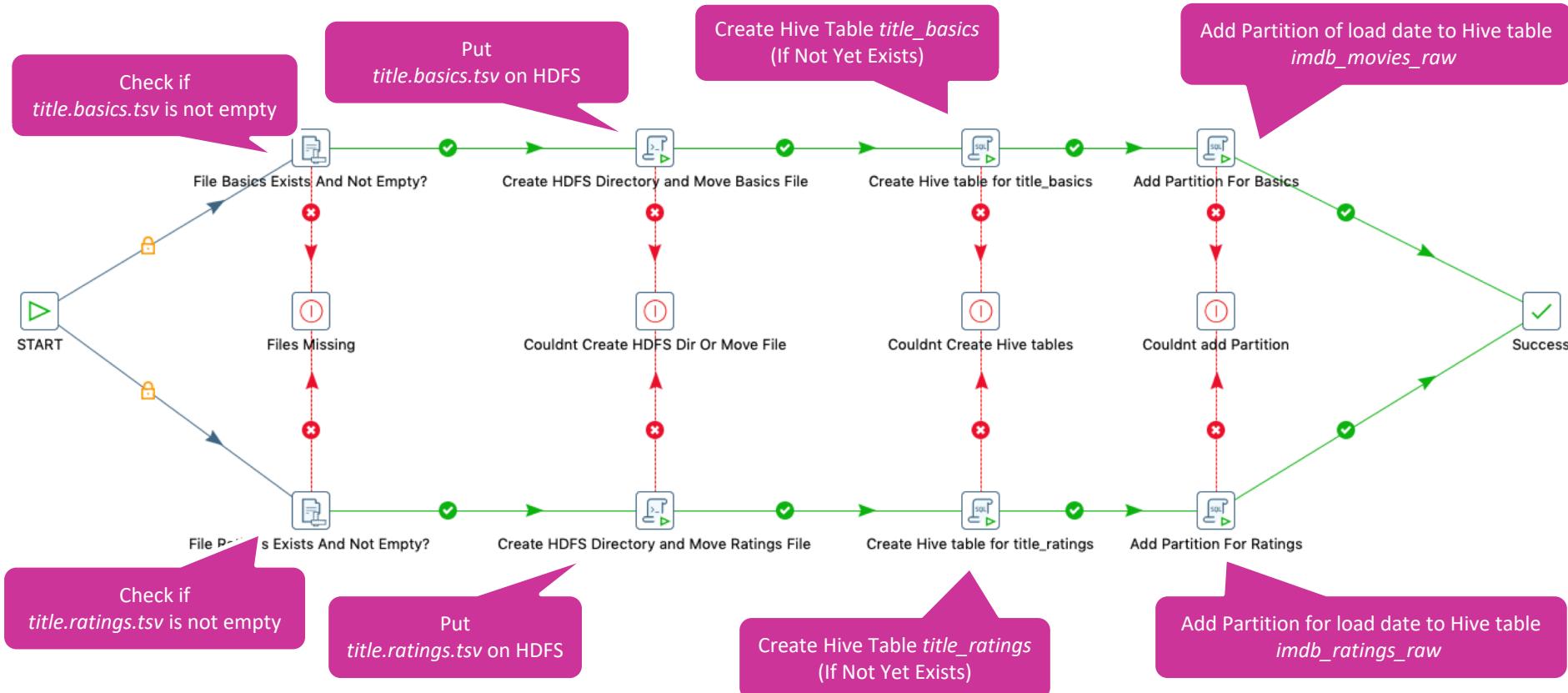
PDI IMDb Import – Main Job (*imdb_main.kjb*)



PDI IMDb Import – *imdb_download.kjb*



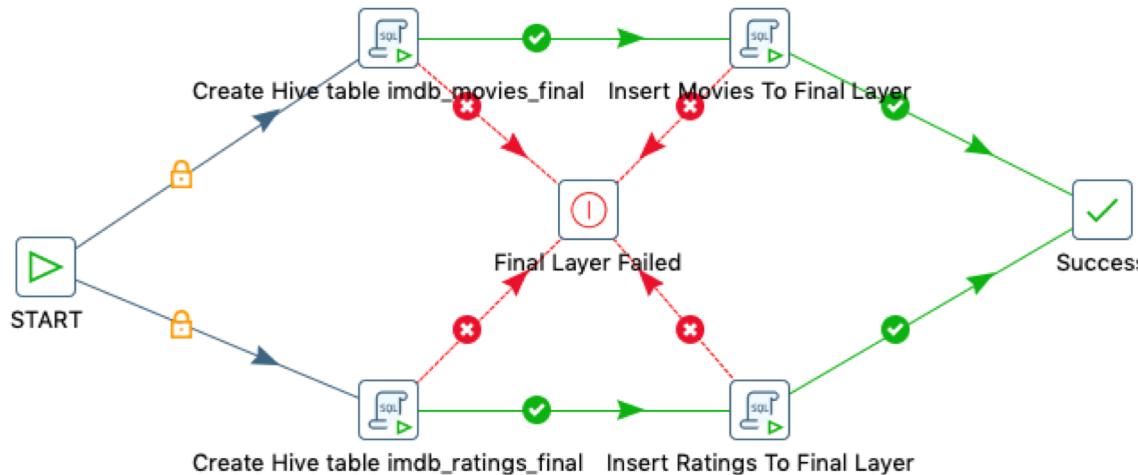
PDI IMDb Import – *imdb_import_to_hdfs_raw.kjb*



PDI IMDb Import – *raw_to_final.kjb*

Create Hive Table *imdb_movies*
(If Not Yet Exists)

Fill table *imdb_movies* by using:
- raw layer table *title_basics* and
- **Dynamic** partitioning



Create Hive Table *imdb_ratings*
(If Not Yet Exists)

Fill table *imdb_ratings* by using:
- raw layer table *title_ratings* and
- **Static** partitioning

PDI IMDb Import – Execution

1. Execute using kitchen.sh on ETL Server:

```
/home/pentaho/pentaho/data-integration/kitchen.sh -file=/home/pentaho/pdi_jobs/imdb_main.kjb  
-param:load_year=2019 -param:load_month=11 -param:load_day=3
```

A Job like this could be scheduled e.g. by cron to run on a daily basis, receiving parameters:

- load_day
- load_month
- load_year

... from crontab job to run every day 06:30 a.m. in the morning and logging everything:

```
30 6 * * * /home/pentaho/pentaho/data-integration/kitchen.sh -file=/home/pentaho/pdi_jobs/imdb_main.kjb  
-param:load_year=`date --date=-1days +\%Y` -param:load_month=`date --date=-1days +\%m`  
-param:load_day=`date --date=-1days +\%d` >> ~/log/imdb_log_$(date +\%Y\%m\%d).log
```





Exercises I

Use Pentaho Data Integration to solve exercises
based on IMDb data



Pentaho Data Integration Exercises – IMDB

1. Execute Tasks of previous HandsOn Slides
2. Use PDI and previous **Jobs&Transformations** to do following changes:
 - a) **Extend job** *imdb_download.kjb* to also download ***name.basics.tsv.gz***
 - b) **Extend job** *imdb_import_to_hdfs_raw.kjb* to also import ***name.basics.tsv*** to HDFS raw layer.
 - c) **Create Hive table** ***name_basics*** for ***name.basics.tsv*** in raw layer.
Table should be partitioned by year, month and day of load date like the other tables.
 - d) **Create table** ***imdb_actors*** and **extend job** ***raw_to_final.kjb*** to also fill table ***imdb_actors_final*** using:
 - data of table ***name_basics*** and
 - **partition table** by column ***partition_is_alive*** containing „alive“ or „dead“, whether the actor is alive or dead.



Pentaho Data Integration Exercises – IMDB

3. Run main workflow job **imdb_main.kjb** using:

```
/home/pentaho/pentaho/data-integration/kitchen.sh -file=/home/pentaho/pdi_jobs/imdb_main.kjb  
-param:load_year=2019 -param:load_month=11 -param:load_day=3
```

Break

TIME FOR
A
BREAK



HandsOn – Apache Airflow

A quick Introduction to ETL Workflow with
Apache Airflow

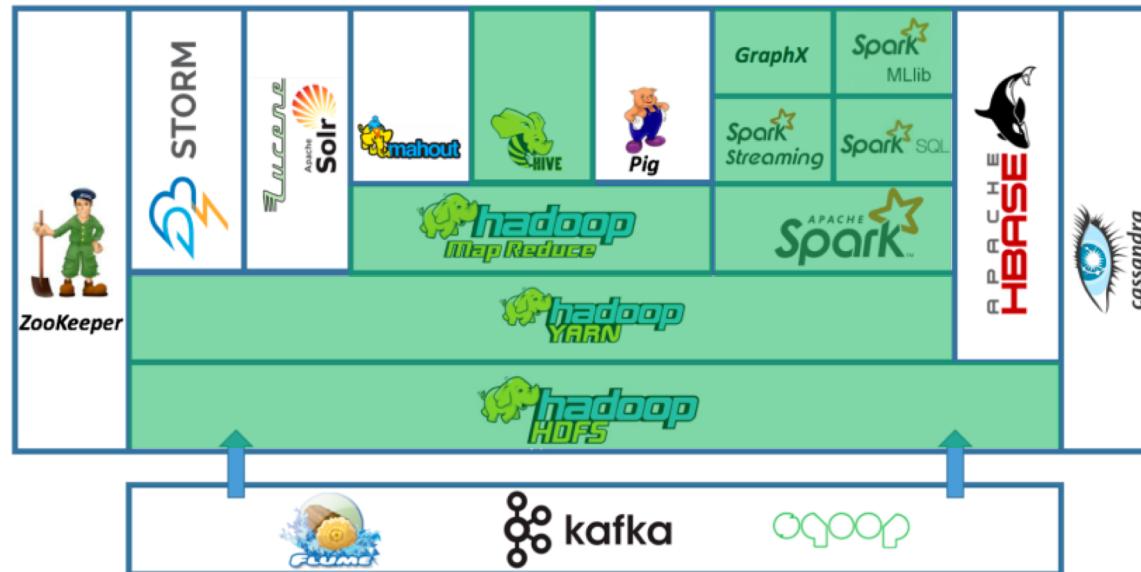


www.marcel-mittelstaedt.com

The Hadoop Ecosystem

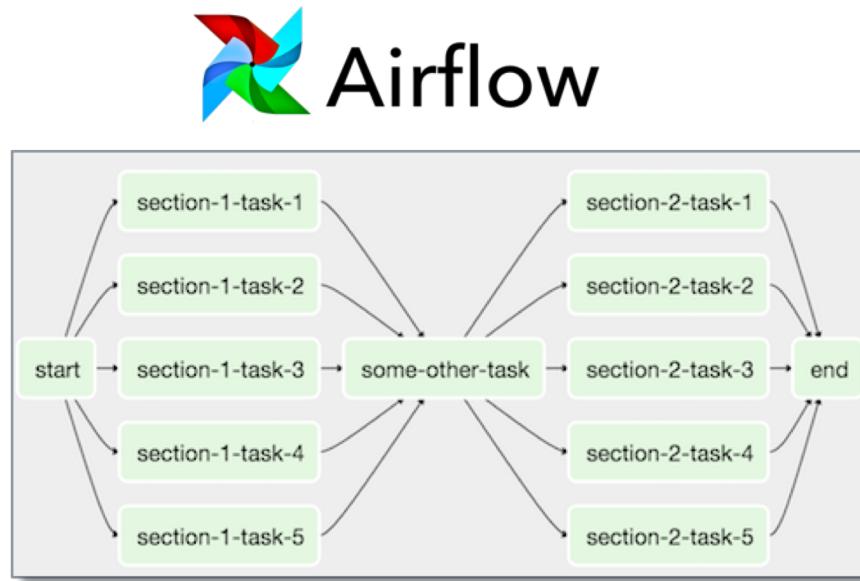


Today's
(exercise) focus



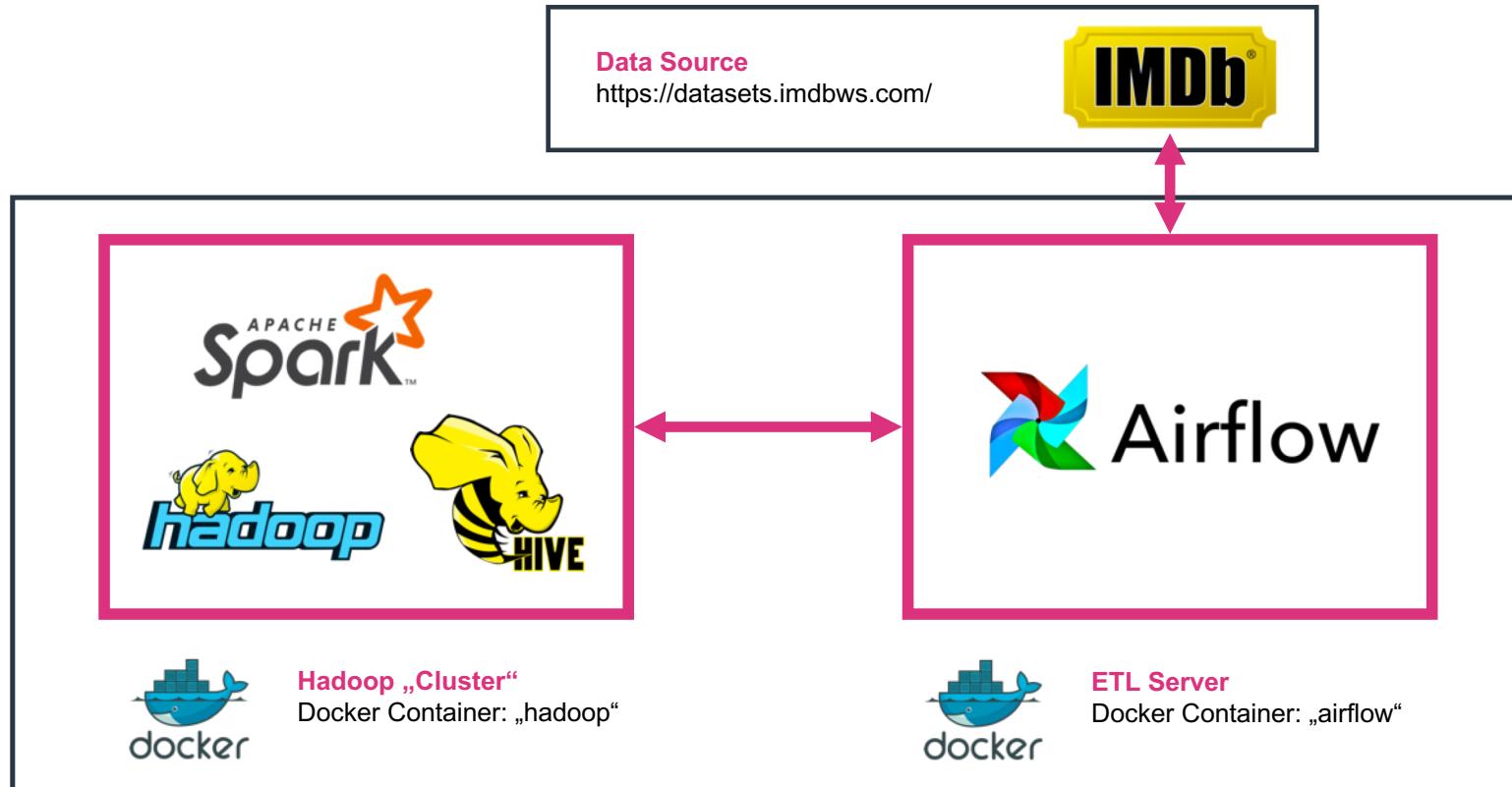
Airflow

- Apache Open Source Project
- Model Task (e.g. ETL) Workflows
- Python Code Base
- Scheduling, Queues and Pools
- Cluster Ready
- Web UI
- **DAG = Directed Acyclic Graph**



→ a collection of all the tasks you want to run, organized in a way that reflects their relationships and dependencies.

What do we want to do?



Remove Previously created Docker Container

1. Stop and Remove Images:

(This will delete all files you created within previous exercise.
Save them somewhere outside the docker container, if you haven't done yet.)

```
docker stop hadoop  
docker rm hadoop
```

```
docker stop pentaho  
docker rm pentaho
```



Start Hadoop/Hive/Spark Docker Container

1. Pull Docker Image:

```
docker pull marcelmittelstaedt/spark_base:latest
```

2. Start Docker Image:

```
docker run -dit --name hadoop \
-p 8088:8088 -p 9870:9870 -p 9864:9864 -p 10000:10000 \
-p 8032:8032 -p 8030:8030 -p 8031:8031 -p 9000:9000 \
-p 8888:8888 --net bigdatanet \
marcelmittelstaedt/spark_base:latest
```

3. Wait till first Container Initialization finished:

```
docker logs hadoop

[...]
Stopping nodemanagers
Stopping resourcemanager
Container Startup finished.
```



Start Hadoop/Hive/Spark Docker Container

4. Get into Docker container:

```
docker exec -it hadoop bash
```

5. Switch to hadoop user:

```
sudo su hadoop
```

```
cd
```

6. Start Hadoop Cluster:

```
start-all.sh
```

7. Start HiveServer2:

```
hiveserver2
```



Start Hadoop/Hive/Spark Docker Container

8. Start HiveServer2:

```
hive/bin/hiveserver2

2018-10-02 16:19:08: Starting HiveServer2
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/home/hadoop/hive/lib/log4j-slf4j-impl-2.10.0.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/home/hadoop/hadoop/share/hadoop/common/lib/slf4j-log4j12-1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.apache.logging.slf4j.Log4jLoggerFactory]
Hive Session ID = b8d1efb3-fc8c-4ec8-bdf0-6a9a41e2ddaa
Hive Session ID = 32503981-a5fd-497e-b887-faf3ec1e686e
Hive Session ID = 00f7eab4-5a29-4ce4-ad97-e90904d9206f
Hive Session ID = 100e54c5-14c6-4acc-b398-040152b08ebf
[...]
```



Start ETL (Airflow) Docker Container

1. Pull Docker Image:

```
docker pull marcelmittelstaedt/airflow:latest
```

2. Start Docker Image:

```
docker run -dit --name airflow \
-p 8080:8080 \
--net bigdatanet \
marcelmittelstaedt/airflow:latest
```

3. Wait till first Container Initialization finished:

```
docker logs airflow

[...]
Successfully added `conn_id`=spark : spark://:@yarn:

Container Startup finished.
```



Start ETL (Airflow) Docker Container

4. Get into Docker container:

```
docker exec -it airflow bash
```

5. Switch to airflow user:

```
sudo su airflow
```

```
cd
```

Exercises Preparation II

Airflow First Steps/Dag



www.marcel-mittelstaedt.com

Spoon Interface

Airflow Landing Page <http://xxx.xxx.xxx.xxx:8080/admin/>

Quick Links to e.g.:

- Trigger Dag
- View Dag
- View Execution Logs
- View Code

- ...

The screenshot shows the Airflow 'DAGs' page. At the top, there's a navigation bar with links for Airflow, DAGs, Data Profiling, Browse, Admin, Docs, and About. On the right side of the header, it shows the date and time as 2019-11-02 20:43:01 UTC. Below the header is a search bar labeled 'Search:'.

DAGs (highlighted by a pink speech bubble)

DAG scheduled? (on/off) (highlighted by a pink speech bubble) - A button labeled 'On' with a blue icon.

Schedule Time (Cron) (highlighted by a pink speech bubble) - A field showing '56 18 * * *'.

Recent Executions (highlighted by a pink speech bubble) - A circular progress bar with 12 green segments, 2 grey segments, and 5 blue segments.

Last Execution of DAG (highlighted by a pink speech bubble) - A timestamp: 2019-11-02 18:56.

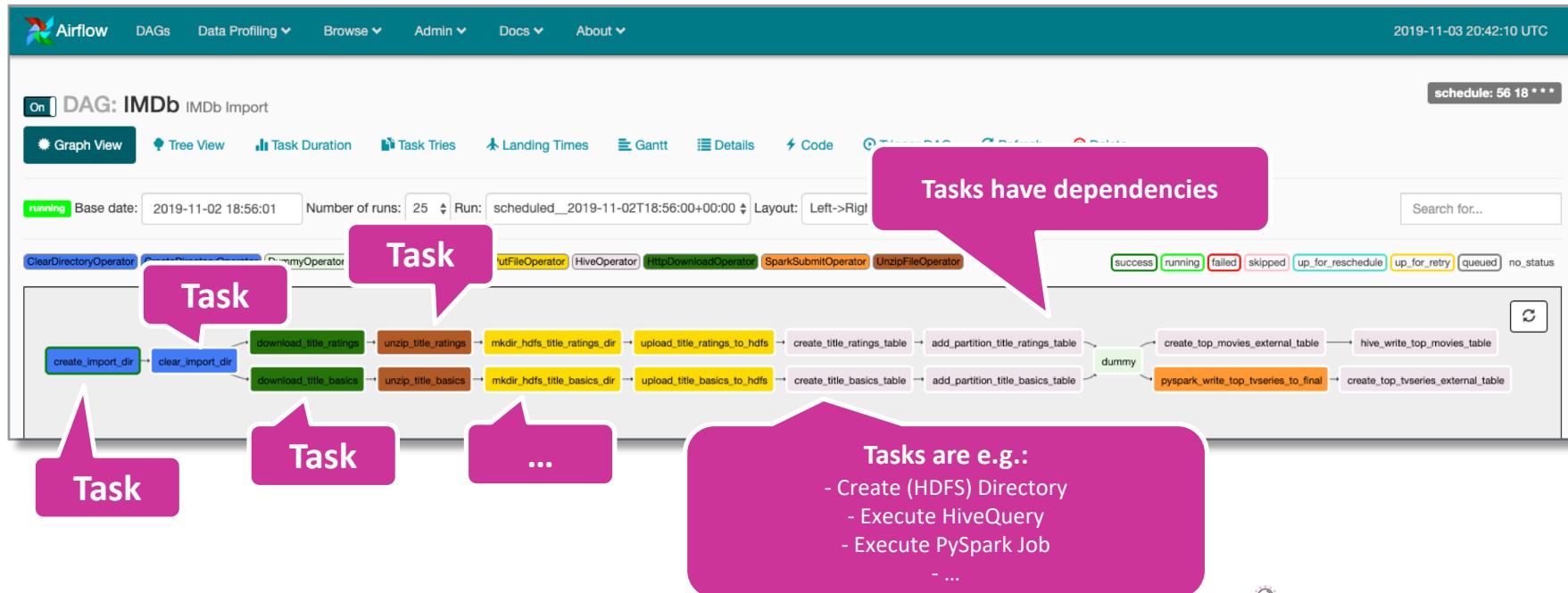
	DAG	Schedule	Owner	Recent Tasks	Last Run	DAG Runs	Links
On	IMDb	56 18 * * *	airflow	12 2 5	2019-11-02 18:56	1	...

Showing 1 to 1 of 1 entries



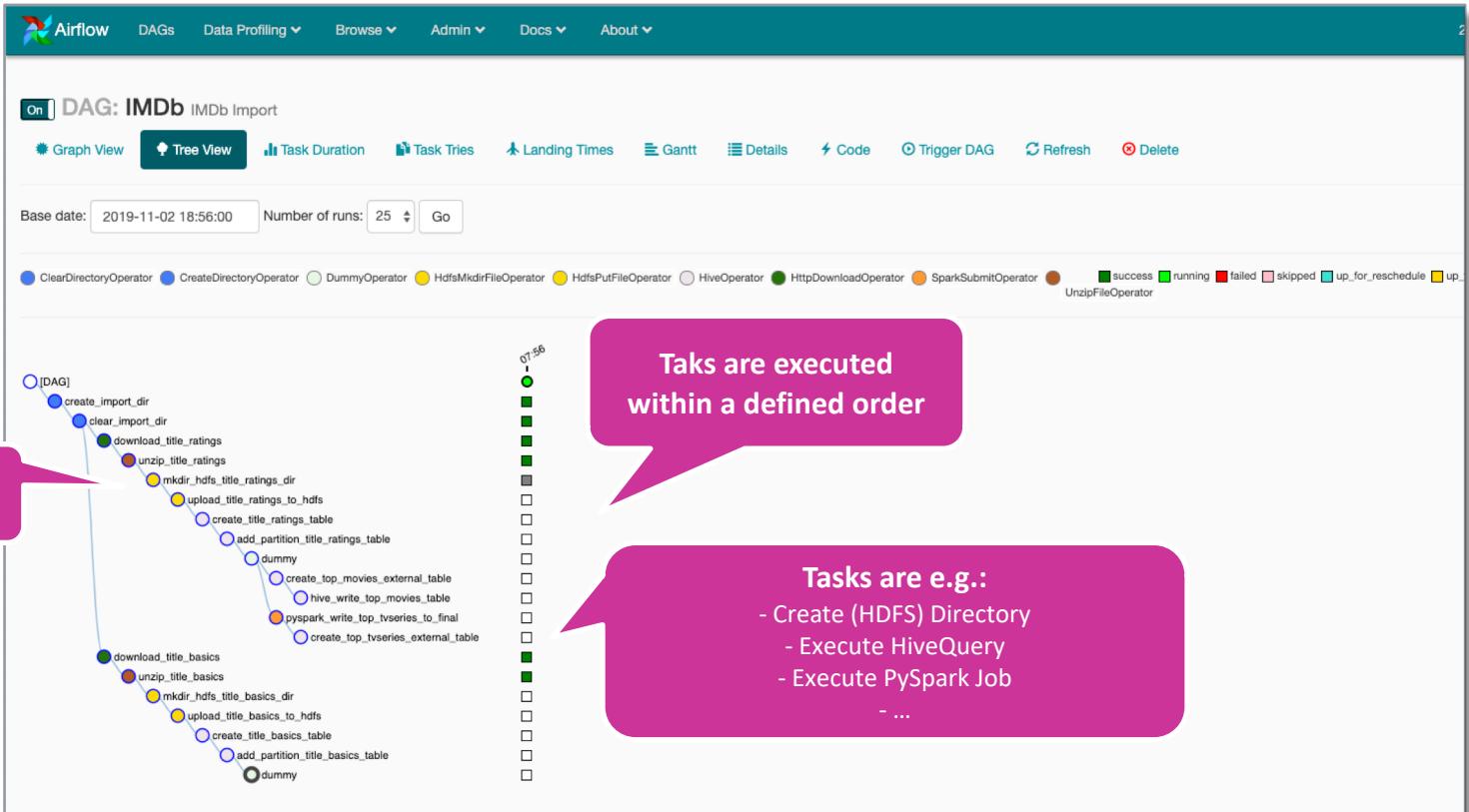
Spoon Interface

Graph View:



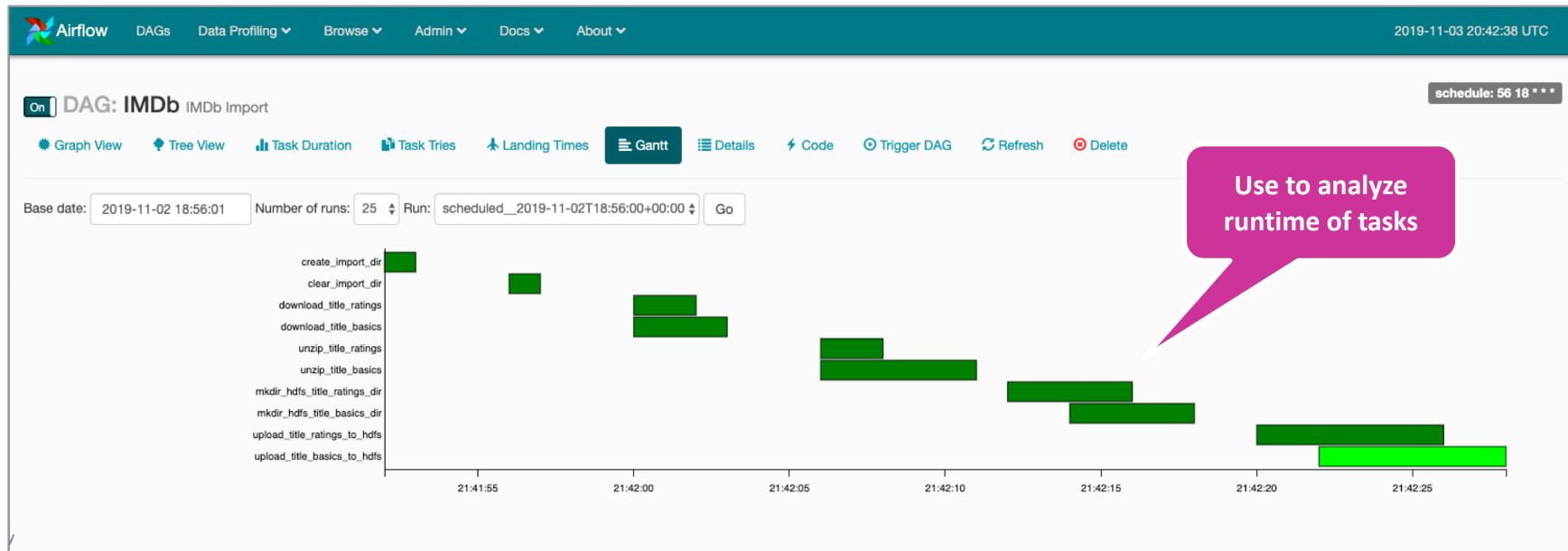
Spoon Interface

Tree View:



Spoon Interface

Gantt View:



Spoon Interface

Logs View:

DAG: IMDb IMDb Import

Task Instance: unzip_title_ratings 2019-11-02 18:56:00

Log

Log by attempts

1

Toggle wrap Jump to end

```
*** Reading local file: /home/airflow/airflow/logs/IMDb/unzip_title_ratings/2019-11-02T18:56:00+00:00/1.log
[2019-11-03 20:42:06,214] {taskinstance.py:630} INFO - Dependencies all met for <TaskInstance: IMDb.unzip_title_ratings 2019-11-02T18:56:00+00:00 [queued]>
[2019-11-03 20:42:06,249] {taskinstance.py:630} INFO - Dependencies all met for <TaskInstance: IMDb.unzip_title_ratings 2019-11-02T18:56:00+00:00 [queued]>
[2019-11-03 20:42:06,249] {taskinstance.py:841} INFO -
```

```
[2019-11-03 20:42:06,249] {taskinstance.py:842} INFO - Starting attempt 1 of 1
[2019-11-03 20:42:06,249] {taskinstance.py:843} INFO -
```

```
[2019-11-03 20:42:06,270] {taskinstance.py:862} INFO - Executing <Task(UnzipFileOperator): unzip_title_ratings> on 2019-11-02T18:56:00+00:00
[2019-11-03 20:42:06,271] {base_task_runner.py:133} INFO - Running: ['airflow', 'run', 'IMDb', 'unzip_title_ratings', '2019-11-02T18:56:00+00:00', '--job_id', '6', '--pool', 'default_pool', '--raw', '-sd', 'DAGS_FC'
[2019-11-03 20:42:07,151] {base_task_runner.py:115} INFO - Job 6: Subtask unzip_title_ratings [2019-11-03 20:42:07,151] {settings.py:252} INFO - settings.configure_orm(): Using pool settings. pool_size=5, max_overf
[2019-11-03 20:42:07,185] {base_task_runner.py:115} INFO - Job 6: Subtask unzip_title_ratings /home/airflow/.local/lib/python3.6/site-packages/psycopg2/_init__.py:144: UserWarning: The psycopg2 wheel package will
[2019-11-03 20:42:07,185] {base_task_runner.py:115} INFO - Job 6: Subtask unzip_title_ratings """
[2019-11-03 20:42:07,944] {base_task_runner.py:115} INFO - Job 6: Subtask unzip_title_ratings [2019-11-03 20:42:07,942] {__init__.py:51} INFO - Using executor LocalExecutor
[2019-11-03 20:42:07,944] {base_task_runner.py:115} INFO - Job 6: Subtask unzip_title_ratings [2019-11-03 20:42:07,944] {dagbag.py:92} INFO - Filling up the DagBag from /home/airflow/airflow/dags/imdb.py
[2019-11-03 20:42:07,994] {base_task_runner.py:115} INFO - Job 6: Subtask unzip_title_ratings [2019-11-03 20:42:07,994] {cli.py:545} INFO - Running <TaskInstance: IMDb.unzip_title_ratings 2019-11-02T18:56:00+00:00
[2019-11-03 20:42:08,029] {zip_file_operator.py:33} INFO - UnzipFileOperator execution started.
[2019-11-03 20:42:08,029] {zip_file_operator.py:35} INFO - Unzipping '/home/airflow/imdb/title.ratings_2019-11-02.tsv.gz' to '/home/airflow/imdb/title.ratings_2019-11-02.tsv'.
[2019-11-03 20:42:08,200] {zip_file_operator.py:40} INFO - UnzipFileOperator done.
[2019-11-03 20:42:11,163] {logging_mixin.py:112} INFO - [2019-11-03 20:42:11,163] {local_task_job.py:124} WARNING - Time since last heartbeat(0.02 s) < heartrate(5.0 s), sleeping for 4.979647 s
[2019-11-03 20:42:16,150] {logging_mixin.py:112} INFO - [2019-11-03 20:42:16,148] {local_task_job.py:103} INFO - Task exited with return code 0
```



Create Simple Example DAG

1. Open Dag File:

```
vi /home/airflow/airflow/dags/example_dag.py
```

```
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime, timedelta

args = {
    'owner': 'airflow'
}

dag = DAG('ExampleDAG', default_args=args, description='Simple Example DAG',
          schedule_interval='56 18 * * *',
          start_date=datetime(2019, 10, 16), catchup=False, max_active_runs=1)
```

```
task_1 = BashOperator(
    task_id='print_date',
    bash_command='date',
    dag=dag)
```

```
task_2 = BashOperator(
    task_id='sleep',
    bash_command='sleep 5',
    retries=3,
    dag=dag)
```

```
task_1 >> task_2
```

Task 1

Task 2

DAG Definition, e.g.

- Name
- Schedule Interval (Cron)
- Description
- Start date
- ...

Task Execution Order

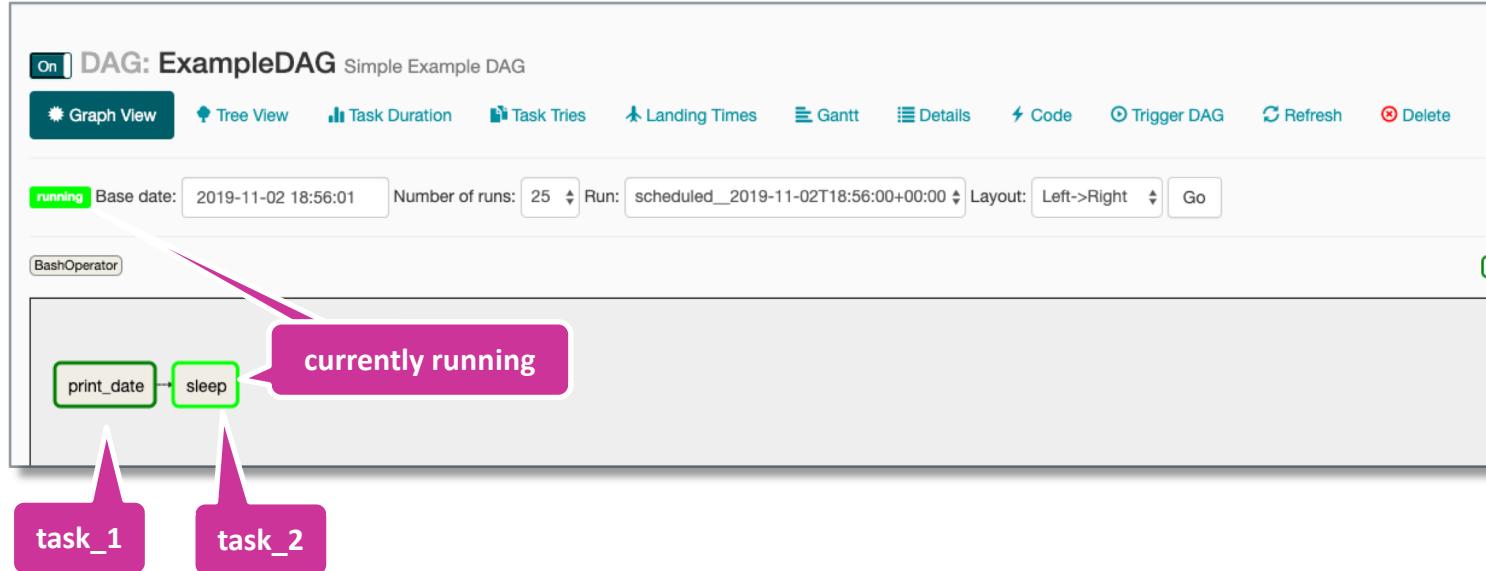
task_2 is dependent on task_1

Spoon Interface

Execute DAG:

Spoon Interface

See executing DAG:



Spoon Interface

View Log of task *print_date*:

2019-11-03 21:11:46 UTC

On [DAG: ExampleDAG

Graph View Tree View Refresh Delete

running Base date: 2019-11-02

BashOperator

print_date → sleep

print_date on 2019-11-02T18:56:00+00:00:00

Task Instance Details Rendered Task Instances View Log

Download Log (by attempts):

1

Run Ignore All Deps Ignore Task Status

Clear Past Future Upstream Downstream Recursive Failed

Mark Failed Past Future Upstream Downstream

Mark Success Past Future Upstream Downstream

Left->Right Go Search for... Up_for_retry queued no_status

Close

See Log

On [DAG: ExampleDAG Simple Example DAG

DAGs Data Profiling Browse Admin Docs About

2019-11-03 21:12:01 UTC

refresh delete

Instance: print_date 2019-11-02 21:08:56:00

task instance details Rendered Template Log XCom

Log by attempts

Toggle wrap Jump to end

*** Local file: /home/airflow/airflow/logs/ExampleDAG/print_date/2019-11-02T18:56:00+00:00/1.log

[2019-11-03 21:10:11,950] {taskinstance.py:630} INFO - Dependencies all met for <TaskInstance: ExampleDAG.print_date> 2019-11-02 [2019-11-03 21:10:11,953] {taskinstance.py:630} INFO - Dependencies all met for <TaskInstance: ExampleDAG.print_date> 2019-11-02 [2019-11-03 21:10:11,954] {taskinstance.py:841} INFO -

[2019-11-03 21:10:11,954] {taskinstance.py:842} INFO - Starting attempt 1 of 1

[2019-11-03 21:10:11,954] {taskinstance.py:843} INFO -

[2019-11-03 21:10:11,969] {taskinstance.py:862} INFO - Executing <Task(BashOperator): print_date> on 2019-11-02T18:56:00+00:00

[2019-11-03 21:10:11,970] {base_task_runner.py:133} INFO - Running: ['airflow', 'run', 'ExampleDAG', 'print_date', '2019-11-02T18:56:00+00:00']

[2019-11-03 21:10:12,835] {base_task_runner.py:115} INFO - Job 21: Subtask print_date [2019-11-03 21:10:12,835] {settings.py:25}

[2019-11-03 21:10:12,860] {base_task_runner.py:115} INFO - Job 21: Subtask print_date /home/airflow/.local/lib/python3.6/site-packages/airflow/operators/bash_operator.py:66: [2019-11-03 21:10:12,860] {base_task_runner.py:115} INFO - Job 21: Subtask print_date ""

[2019-11-03 21:10:13,598] {base_task_runner.py:115} INFO - Job 21: Subtask print_date [2019-11-03 21:10:13,598] {__init__.py:51}

[2019-11-03 21:10:13,598] {base_task_runner.py:115} INFO - Job 21: Subtask print_date [2019-11-03 21:10:13,597] {dabag.py:92}

[2019-11-03 21:10:13,637] {base_task_runner.py:115} INFO - Job 21: Subtask print_date [2019-11-03 21:10:13,636] {cli.py:545} IN

[2019-11-03 21:10:13,668] {bash_operator.py:81} INFO - Tmp dir root location: /tmp

[2019-11-03 21:10:13,668] {bash_operator.py:91} INFO - Exporting the following env vars:

AIRFLOW_CTX_DAG_ID=ExampleDAG

AIRFLOW_CTX_TASK_ID=print_date

AIRFLOW_CTX_EXECUTION_DATE=2019-11-02T18:56:00+00:00

AIRFLOW_CTX_DAG_RUN_ID=scheduled_2019-11-02T18:56:00+00:00

[2019-11-03 21:10:13,668] {bash_operator.py:105} INFO - Temporary script location: /tmp/airflowtmppassn3x9j/print_datev2ze8bz

[2019-11-03 21:10:13,668] {bash_operator.py:114} INFO - Running command: date

[2019-11-03 21:10:13,676] {bash_operator.py:104} INFO - Output:

date: command exited with return code 0

[2019-11-03 21:10:13,676] {bash_operator.py:120} INFO - Task exit status: 0

date bash command of task_1



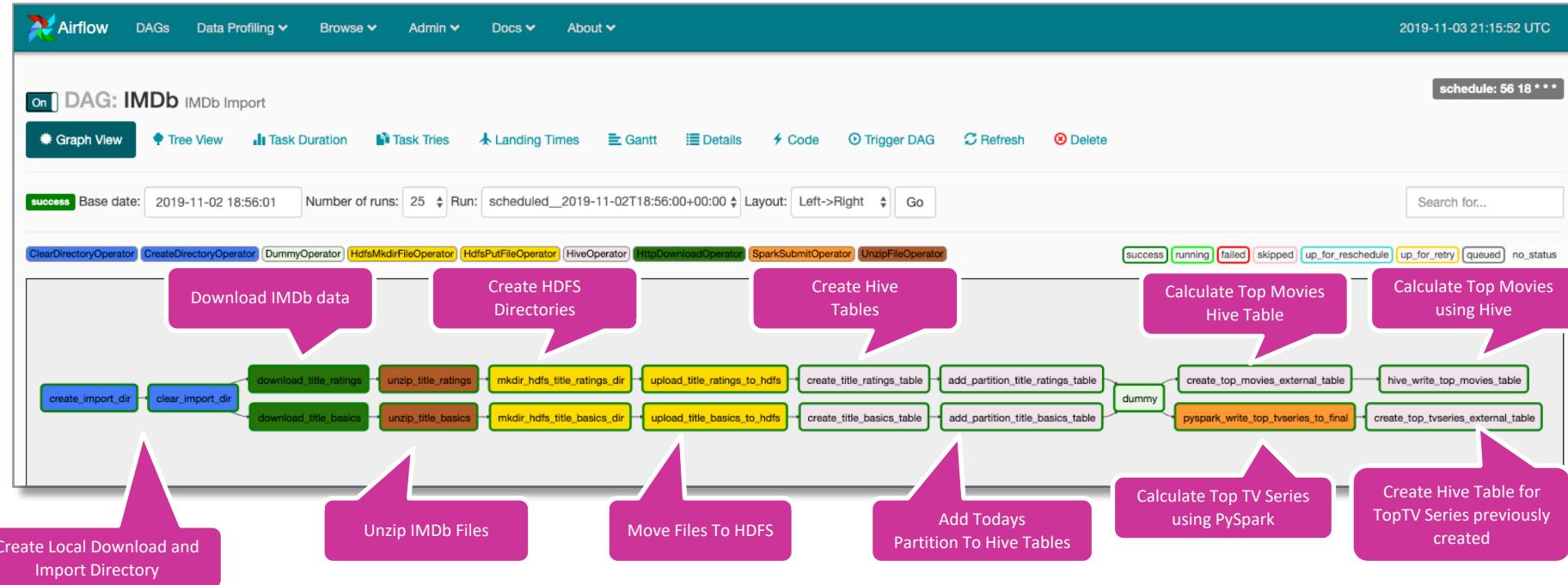
Exercises II

Use Apache Airflow to solve exercises based on
IMDb data



Spoon Interface

See executing DAG:



Pentaho Data Integration Exercises – IMDB

1. Execute IMDb DAG
2. Use [Airflow](#) and previous IMDb [DAG](#) to do following changes (`vi /home/airflow/airflow/dags/imdb.py`):
 - a) Extend Airflow IMDb DAG to also download `name.basics.tsv.gz`
 - b) Extend Airflow IMDb DAG to also import `name.basics.tsv` to HDFS raw layer.
 - c) Create Hive table `name_basics` for `name.basics.tsv` in raw layer within DAG. Table should be partitioned by year, month and day of load date like the other tables.
 - d) Create table `actors` and extend IMDb Airflow DAG to fill table using Hive or PySpark:
 - make use of all columns within table `name_basics`
 - add column `alive` which contains alive if actor is alive or dead if actor is dead
 - add column `age` which contains current age of actor (calculated by using birth and death year)
 - e) Run DAG



Well Done

WE'RE DONE
FOR
...TODAY

