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

*Winter Semester 2025/2026,*
*Cooperative State University Baden-Wuerttemberg*


# Agenda – 27.10.2025

**01** **Presentation and Discussion: Exercise Of Last Lecture**
PySpark, Jupyter, Spark DataFrames and Partitioning on IMDb dataset

**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** **HandsOn – Introduction to ETL Workflow Tools (PDI and Airflow)**
Automate the process of downloading, importing, transforming and storing IMDb data within a Hadoop Cluster.

**04** **Exercise – Working with Apache Airflow on Hadoop, Hive and Spark**
Automate the process of downloading, importing, transforming and storing IMDb data within a Hadoop Cluster.

# Schedule

| | | | *Lecture Topic* | *HandsOn* |
|---|---|---|---|---|
| 29.09.2025 | 13:15-15:45 | Ro. N/A | About This Lecture, Introduction to Big Data | Setup Google Cloud, Create Own Hadoop Cluster and Run MapReduce |
| 06.10.2025 | 13:15-15:45 | Ro. N/A | (Non-)Functional Requirements Of Distributed Data-Systems, Data Models and Access | Hive and HiveQL |
| 13.10.2025 | 13:15-15:45 | Ro. N/A | Challenges Of Distributed Data Systems: Partitioning | HiveQL via JDBC, Data Partitioning (with HDFS and Hive) |
| 20.10.2025 | 13:15-15:45 | Ro. N/A | Challenges Of Distributed Data Systems: Replication | Spark, Scala, PySpark and Jupyter Notebooks |
| **27.10.2025** | **13:15-15:45** | **Ro. N/A** | **ETL Workflow and Automation & Batch and Stream Processing** | Airflow |
| 03.11.2025 | 13:15-15:45 | Ro. N/A | Practical Exam | Work On Practical Exam |
| 10.11.2025 | 13:15-15:45 | Ro. N/A | Practical Exam | Work On Practical Exam |
| 17.11.2025 | 13:15-15:45 | Ro. N/A | Practical Exam | Work On Practical Exam |
| **24.11.2025** | **13:15-15:45** | **Ro. N/A** | **Practical Exam Presentation** | |
| **01.12.2025** | **13:15-15:45** | **Ro. N/A** | **Practical Exam Presentation** | |

# Solution – Exercise 04

Spark, PySpark, Jupyter

# Solution

**Prerequisites:**

- Start Gcloud instance
- Pull and start Docker image (`marcelmittelstaedt/spark_base:latest`)

- Start HDFS and YARN
- Start Jupyter Notebook
- Execute all preparation and example tasks of previous HandsOn slides of last lecture

See:

https://github.com/marcelmittelstaedt/BigData/tree/master/solutions/winter_semester_2025-2026/04_spark_pyspark_jupyter/Solutions.html

…for complete solution (Jupyter Notebook).

# Solution

1.) Start Spark Session:

```
# Import Spark Libraries
import findspark, os
findspark.init('/home/hadoop/spark')
from pyspark.sql import SparkSession

# Initialize Spark Session
spark = SparkSession.builder \
    .master("yarn") \
    .appName("Jupyter/PySpark Exercises") \
    .enableHiveSupport() \
    .getOrCreate()
```

# Solution

2. Create External Spark Table **`title_ratings`** on HDFS containing data of IMDb file *title.ratings.tsv*

```
# EXERCISE 2) Create External Spark Table title_ratings on HDFS containing data of IMDb file title.ratings.tsv

# Read IMDb title ratings CSV file from HDFS
df_title_ratings = spark.read \
    .format('csv') \
    .options(header='true', delimiter='\t', nullValue='null', inferSchema='true') \
    .load('/user/hadoop/imdb/title_ratings/*.tsv')

# Save Dataframe back to HDFS (partitioned) as EXTERNAL TABLE and Parquet files
df_title_ratings.write \
    .format("parquet") \
    .mode("overwrite") \
    .option('path', '/user/hadoop/imdb/title_ratings_table') \
    .saveAsTable('default.title_ratings')

# Check Results:
spark.sql('SELECT * FROM default.title_ratings').show(3)
```

```
+---------+-------------+--------+
|   tconst|averageRating|numVotes|
+---------+-------------+--------+
|tt0000001|          5.7|    1685|
|tt0000002|          6.0|     208|
|tt0000003|          6.5|    1425|
+---------+-------------+--------+
only showing top 3 rows
```

# Solution

3. Create External Spark Table **name_basics** on HDFS containing data of IMDb file *name.basics.tsv*

```
# EXERCISE 3) Create External Spark Table name_basics on HDFS containing data of IMDb file name.basics.tsv

# Read IMDb name basics CSV file from HDFS
df_name_basics = spark.read \
    .format('csv') \
    .options(header='true', delimiter='\t', nullValue='null', inferSchema='true') \
    .load('/user/hadoop/imdb/name_basics/*.tsv')

# Save Dataframe back to HDFS (partitioned) as EXTERNAL TABLE and Parquet files
df_name_basics.write \
    .format("parquet") \
    .mode("overwrite") \
    .option('path', '/user/hadoop/imdb/name_basics_table') \
    .saveAsTable('default.name_basics')

# Check Results:
spark.sql('SELECT * FROM default.name_basics').show(3)
```

```
+---------+---------------+---------+---------+--------------------+--------------------+
|   nconst|    primaryName|birthYear|deathYear|   primaryProfession|      knownForTitles|
+---------+---------------+---------+---------+--------------------+--------------------+
|nm2511361|    Shane Vahey|       \N|       \N|writer,editor,pro...|tt2261585,tt01922...|
|nm2511363|Adolf Seilacher|       \N|       \N|                null|                  \N|
|nm2511364|   Nora Brennan|       \N|       \N|casting_departmen...|tt4029524,tt77364...|
+---------+---------------+---------+---------+--------------------+--------------------+
only showing top 3 rows
```

# Solution

4.a) How many **movies** and how many **TV series** are within the IMDB dataset?

```
# EXERCISE 4a) How many movies and how many TV series are within the IMDB dataset?

# Programmatical Approach
from pyspark.sql.functions import col
df = spark.table('default.title_basics_partitioned') \
    .where(col('titleType').isin(['movie', 'tvSeries'])) \
    .groupBy('titleType') \
    .count()

df.show(100)
```

```
+---------+------+
|titleType| count|
+---------+------+
| tvSeries|202321|
|    movie|569437|
+---------+------+
```

```
# EXERCISE 4a) How many movies and how many TV series are within the IMDB dataset?

# SQL Approach
df = spark.sql('''
    SELECT titleType, count(*)
    FROM default.title_basics_partitioned
    WHERE titleType IN ("movie", "tvSeries")
    GROUP BY titleType
    ''')

df.show(100)
```

```
+---------+--------+
|titleType|count(1)|
+---------+--------+
| tvSeries|  202321|
|    movie|  569437|
+---------+--------+
```

# Solution

4.b) Who is the **youngest** actor/writer/… within the dataset?

```
# EXERCISE 4b) Who is the youngest actor/writer/… within the dataset?

# Programmatical Approach
from pyspark.sql.functions import col
df = spark.table('default.name_basics') \
    .where(col('birthYear') != '\\N') \
    .sort(col('birthYear').desc())

df.show(3)
```

```
+----------+--------------------+---------+---------+-----------------+-------------------+
|    nconst|         primaryName|birthYear|deathYear|primaryProfession|     knownForTitles|
+----------+--------------------+---------+---------+-----------------+-------------------+
| nm0894719|        Sarah Vernon|     2021|       \N|          actress|tt0084987,tt0090499|
|nm11763191|          Win Wilson|     2020|       \N|             null|                 \N|
|nm12122609|Adam James Sanderson|     2020|       \N|            actor|         tt12668798|
+----------+--------------------+---------+---------+-----------------+-------------------+
only showing top 3 rows
```

```
# EXERCISE 4b) Who is the youngest actor/writer/… within the dataset?

# SQL Approach
df = spark.sql(r"SELECT * FROM default.name_basics WHERE birthYear <> '\\N' ORDER BY birthYear DESC")

df.show(3)
```

```
+----------+--------------------+---------+---------+-----------------+-------------------+
|    nconst|         primaryName|birthYear|deathYear|primaryProfession|     knownForTitles|
+----------+--------------------+---------+---------+-----------------+-------------------+
| nm0894719|        Sarah Vernon|     2021|       \N|          actress|tt0084987,tt0090499|
|nm11763191|          Win Wilson|     2020|       \N|             null|                 \N|
|nm12122609|Adam James Sanderson|     2020|       \N|            actor|         tt12668798|
+----------+--------------------+---------+---------+-----------------+-------------------+
only showing top 3 rows
```

# Solution

4.c) Create a list (`tconst, original_title, start_year, average_rating, num_votes`) of movies which are:

- equal or newer than year 2010
- have an average rating equal or better than 8,1
- have been voted more than 100.000 times

```python
# EXERCISE 4c) Create a list (tconst, original_title, start_year, average_rating, num_votes) of movies which are:
#  - equal or newer than year 2010
#  - have an average rating equal or better than 8,1
#  - have been voted more than 100.000 times

# Programmatical Approach
from pyspark.sql.functions import col
df_title_basics = spark.table('default.title_basics_partitioned')
df_title_ratings = spark.table('default.title_ratings')

# JOIN Data Frames
joined_df = df_title_basics.join(df_title_ratings, ['tconst'])

# Filter DF
df = joined_df \
    .where(col('startYear') >= '2010') \
    .where(col('averageRating') > 8.1) \
    .where(col('numVotes') > 100000) \
    .select('tconst', 'originalTitle', 'startYear', 'averageRating', 'numVotes')

# Show Result
df.show(10, False)
```

```
+---------+--------------------------------+---------+-------------+--------+
|tconst   |originalTitle                   |startYear|averageRating|numVotes|
+---------+--------------------------------+---------+-------------+--------+
|tt7221388|Cobra Kai                       |2018     |8.6          |110286  |
|tt4154756|Avengers: Infinity War          |2018     |8.4          |843065  |
|tt4633694|Spider-Man: Into the Spider-Verse|2018    |8.4          |380545  |
|tt6763664|The Haunting of Hill House      |2018     |8.6          |183333  |
|tt6966692|Green Book                      |2018     |8.2          |384828  |
|tt2380307|Coco                            |2017     |8.4          |389537  |
|tt3647998|Taboo                           |2017     |8.4          |115867  |
|tt3920596|Big Little Lies                 |2017     |8.5          |157469  |
|tt5071412|Ozark                           |2017     |8.4          |189152  |
|tt5290382|Mindhunter                      |2017     |8.6          |218549  |
+---------+--------------------------------+---------+-------------+--------+
only showing top 10 rows
```

# Solution

4.d) Save result of c) as external Spark Table to HDFS.

```
# EXERCISE 4d) Save result of c) as external Spark Table to HDFS.

# Save Dataframe back to HDFS as external table and Parquet files
df.write \
    .format("parquet") \
    .mode("overwrite") \
    .option('path', '/user/hadoop/imdb/top_movies_table') \
    .saveAsTable('default.top_movies')

# Check Result
spark.sql('SELECT * FROM default.top_movies').show(3)
```

```
+---------+--------------------+---------+-------------+--------+
|   tconst|       originalTitle|startYear|averageRating|numVotes|
+---------+--------------------+---------+-------------+--------+
|tt4158110|           Mr. Robot|     2015|          8.5|  334399|
|tt4508902|One Punch Man: Wa...|     2015|          8.8|  117086|
|tt2431438|              Sense8|     2015|          8.3|  139787|
+---------+--------------------+---------+-------------+--------+
only showing top 3 rows
```

# Solution

5. Create a Spark Table **`name_basics_partitioned`**, which:
   - contains all columns of table `name_basics`
   - is partitioned by column `partition_is_alive`, containing:
     - „*alive*“ in case actor is still alive
     - „*dead*“ in case actor is already dead

```
# EXERCISE 5) Create a Spark Table name_basics_partitioned, which:
#   - contains all columns of table name_basics
#   - is partitioned by column partition_is_alive, containing:
#      - „alive“ in case actor is still alive
#      - „dead“ in case actor is already dead

from pyspark.sql.functions import col, when, lit
df = spark.table('default.name_basics')

# Add column 'partition_is_alive'
df_name_basics = df.withColumn('partition_is_alive',
                               when(col('deathYear') == '\\N', lit('alive')).otherwise(lit('dead')))

# Save Dataframe back to HDFS (partitioned) as EXTERNAL TABLE and Parquet files
df_name_basics.repartition('partition_is_alive').write \
    .format("parquet") \
    .mode("overwrite") \
    .option('path', '/user/hadoop/imdb/name_basics_partitioned_table') \
    .partitionBy('partition_is_alive') \
    .saveAsTable('default.name_basics_partitioned')

# Check Results:
spark.sql('SELECT * FROM default.name_basics_partitioned WHERE primaryName = "Heath Ledger"').show(3)
```

```
+---------+------------+---------+---------+--------------------+--------------------+------------------+
|   nconst| primaryName|birthYear|deathYear|   primaryProfession|      knownForTitles|partition_is_alive|
+---------+------------+---------+---------+--------------------+--------------------+------------------+
|nm0005132|Heath Ledger|     1979|     2008|actor,director,so...|tt0147800,tt04685...|              dead|
+---------+------------+---------+---------+--------------------+--------------------+------------------+
```

# Solution

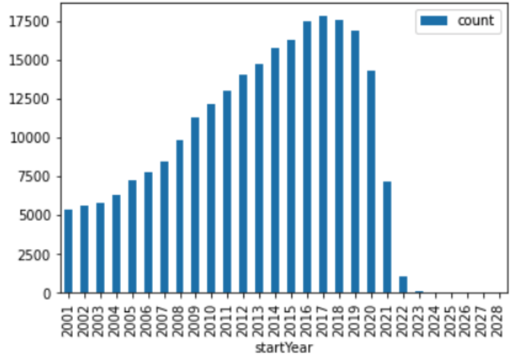
6. Create a partitioned Spark table **`imdb_movies_and_ratings_partitioned`**, which:
   - contains all columns of the two tables `title_basics_partitioned` and `title_ratings` and
   - is partitioned by start year of movie (create and add column `partition_year`).

```
# EXERCISE 6) Create a partitioned Spark table imdb_movies_and_ratings_partitioned, which:
#   - contains all columns of the two tables title_basics_partitioned and title_ratings and
#   - is partitioned by start year of movie (create and add column partition_year).

# Programmatical Approach
from pyspark.sql.functions import col
df_title_basics = spark.table('default.title_basics_partitioned')
df_title_ratings = spark.table('default.title_ratings')

# Join DataFrames
joined_df = df_title_basics.join(df_title_ratings, ['tconst'])

# Add partition column
df = joined_df.withColumn('partition_year', col('startYear'))

# Save DataFrame as external Spark table partitioned by column 'partition_year'
df.repartition('partition_year').write \
    .format("parquet") \
    .mode("overwrite") \
    .option('path', '/user/hadoop/imdb/imdb_movies_and_ratings_partitioned_table') \
    .partitionBy('partition_year') \
    .saveAsTable('default.imdb_movies_and_ratings_partitioned')

# Check Results:
spark.sql('SELECT tconst, titleType, primaryTitle, startYear, endYear, partition_year '
          'FROM default.imdb_movies_and_ratings_partitioned').show(3)
```

```
+----------+---------+--------------------+---------+-------+--------------+
|    tconst|titleType|        primaryTitle|startYear|endYear|partition_year|
+----------+---------+--------------------+---------+-------+--------------+
|tt11115836| tvSeries|Slam Dance: The S...|     2017|   2017|          2017|
|tt11125498|    short|              Snooze|     2017|     \N|          2017|
|tt11125898|tvEpisode|              Ninovo|     2017|     \N|          2017|
+----------+---------+--------------------+---------+-------+--------------+
only showing top 3 rows
```

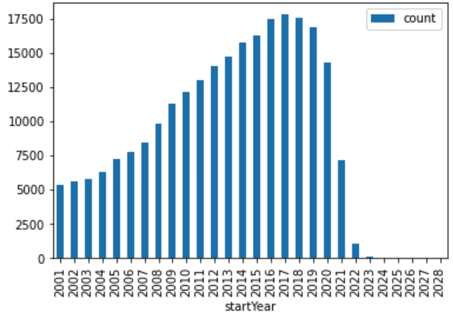
# Solution

7. Create following plot, which visualizes:
   - the amount of movies (type!)
   - per year
   - since 2000




```python
# EXERCISE 7) Create following plot, which visualizes:
#   - the amount of movies (type!)
#   - per year
#   - since 2000

import matplotlib.pyplot as plt
import pandas

# Create DataFrame to be plotted
df = spark.table('default.title_basics_partitioned') \
    .select('startYear', 'titleType') \
    .where(col('startYear') > 2000) \
    .where(col('titleType') == 'movie') \
    .groupBy('startYear') \
    .count() \
    .sort(col('startYear').asc())

# Convert Spark DataFrame to Pandas DataFrame
pandas_df = df.toPandas()

# Plot DataFrame
pandas_df.plot.bar(x='startYear', y='count')
```

```
<AxesSubplot:xlabel='startYear'>
```



Introduction to:

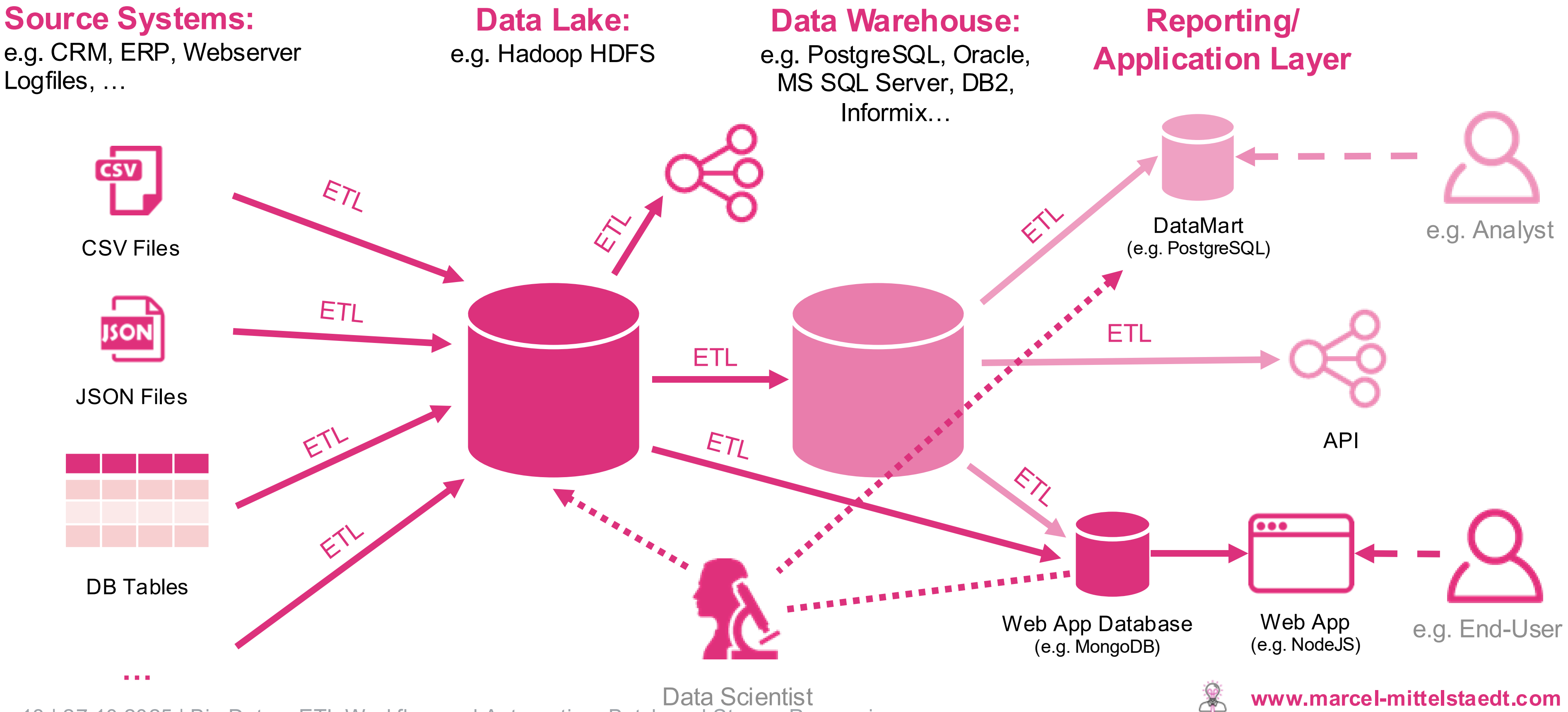
# Batch and Stream Processing

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

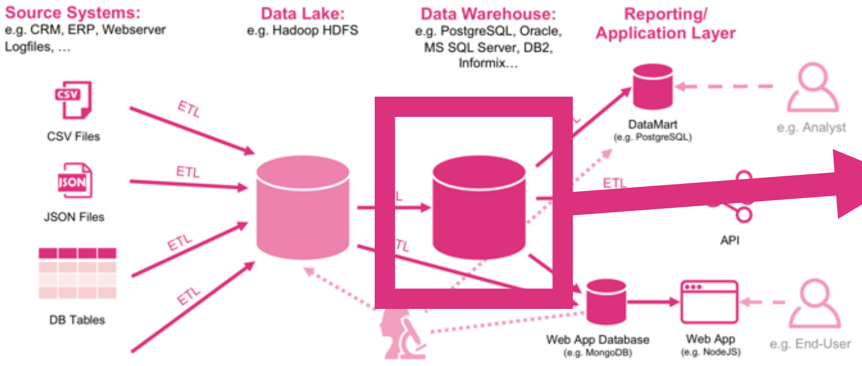


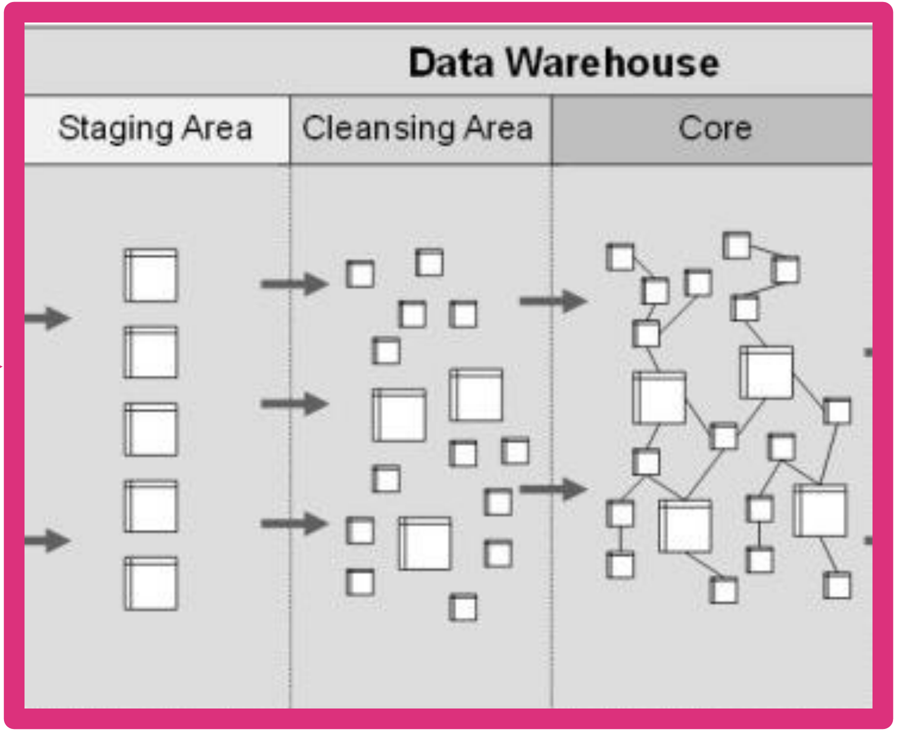
# Batch Processing - ETL

❗

***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 pur poses (e.g. analytical, data science, data-driven applications, …)
- usually not about 100% accuracy

# Distributed Batch Processing

## Distributed Storage

hadoop

cassandra

riak

redis

CouchDB relax

mongoDB®

## Distributed Processing

hadoop Map Reduce

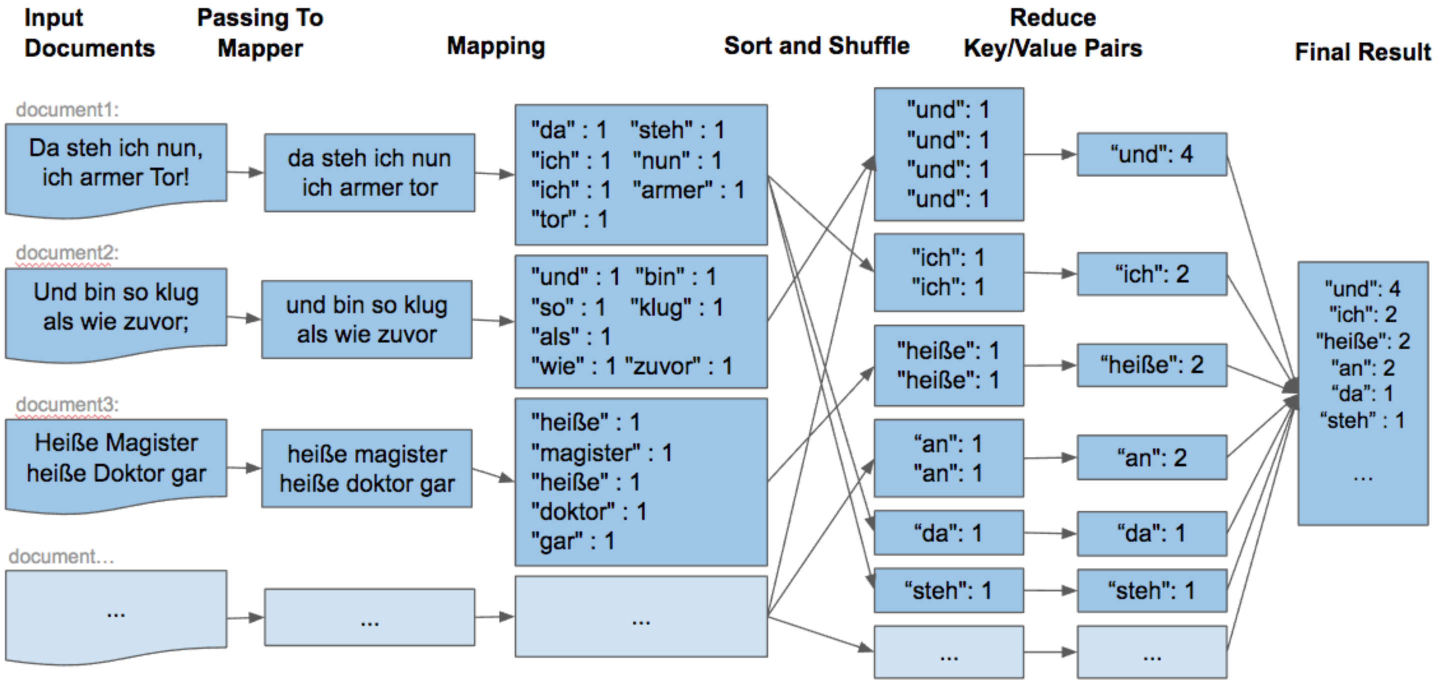
STORM

APACHE Spark™

Flink

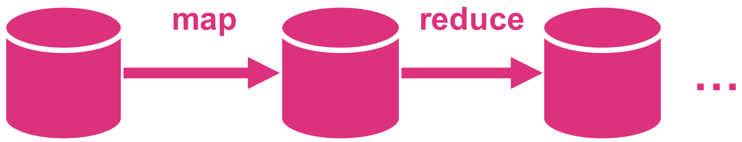
# Batch Processing – **MapReduce**

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



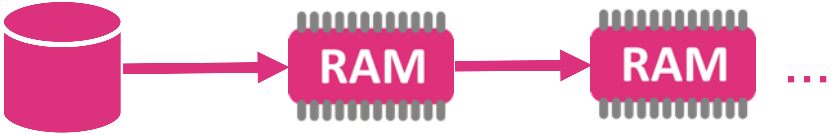
# Batch Processing – MapReduce vs Spark

## Hadoop MapReduce




- Good reliability
- Bad performance

## Spark




- 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 **reliabilty > latency**

- good choice for stream processing:

if **latency > reliability**

# Stream Processing – **Definition**

**Data Stream:**

- is data made available over time in an incemental 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

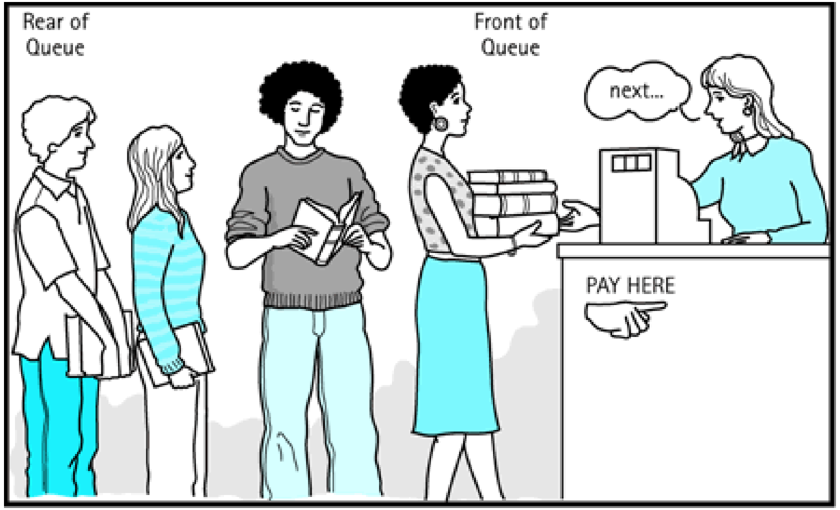
- Location Tracking (e.g. DriveNow, Car2Go, Google Maps…)

- …

# Message Broker/Queues

## Why:

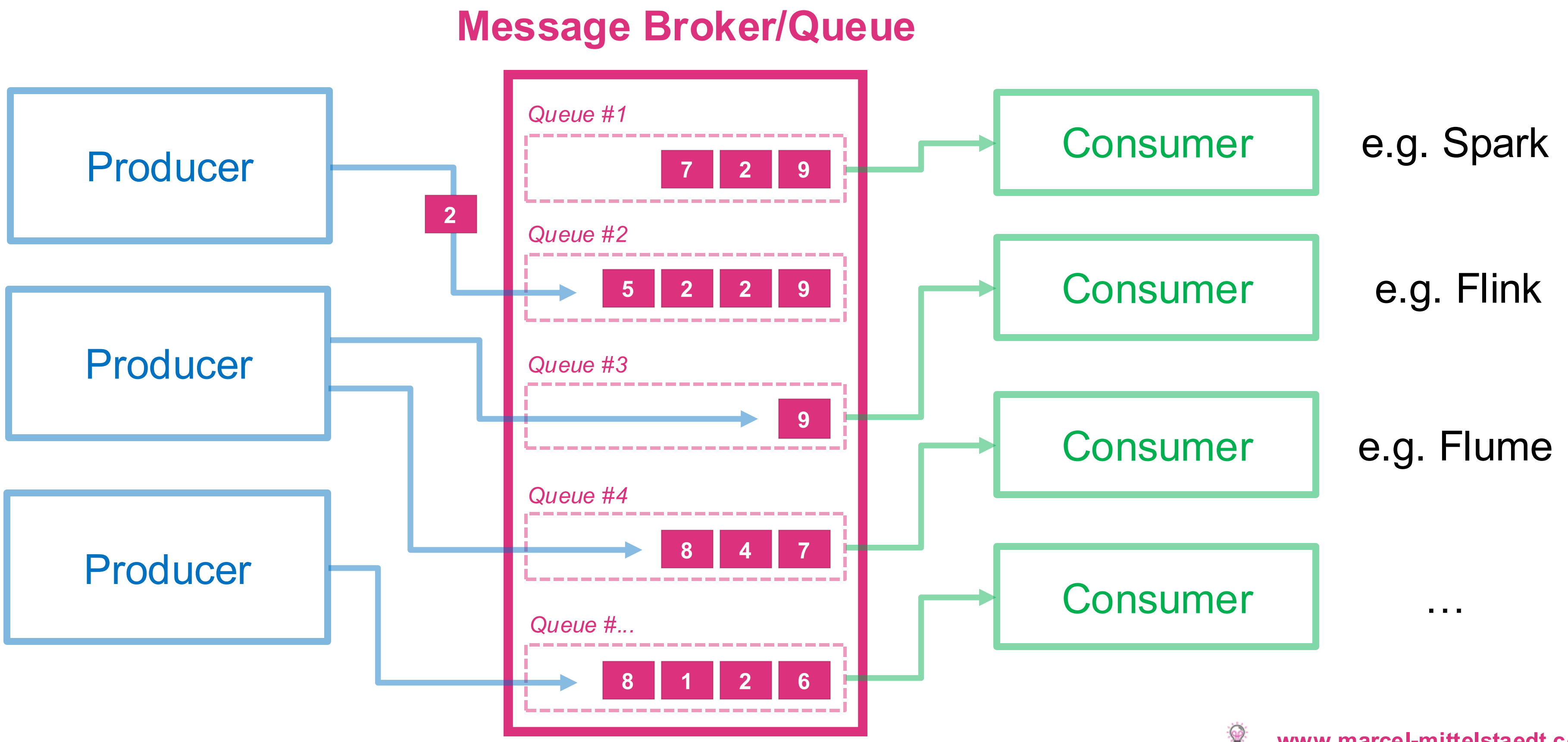
- decouples Producer and Receiver
- maintains queues for different topics
- temporarily persists messages
- Notifies subscribers on new messages
- Micro Services, IoT…




## Examples:

- RabbitMQ,
- ZeroMQ,
- Kafka,
- ActiveMQ,
- Kestrel,
- …

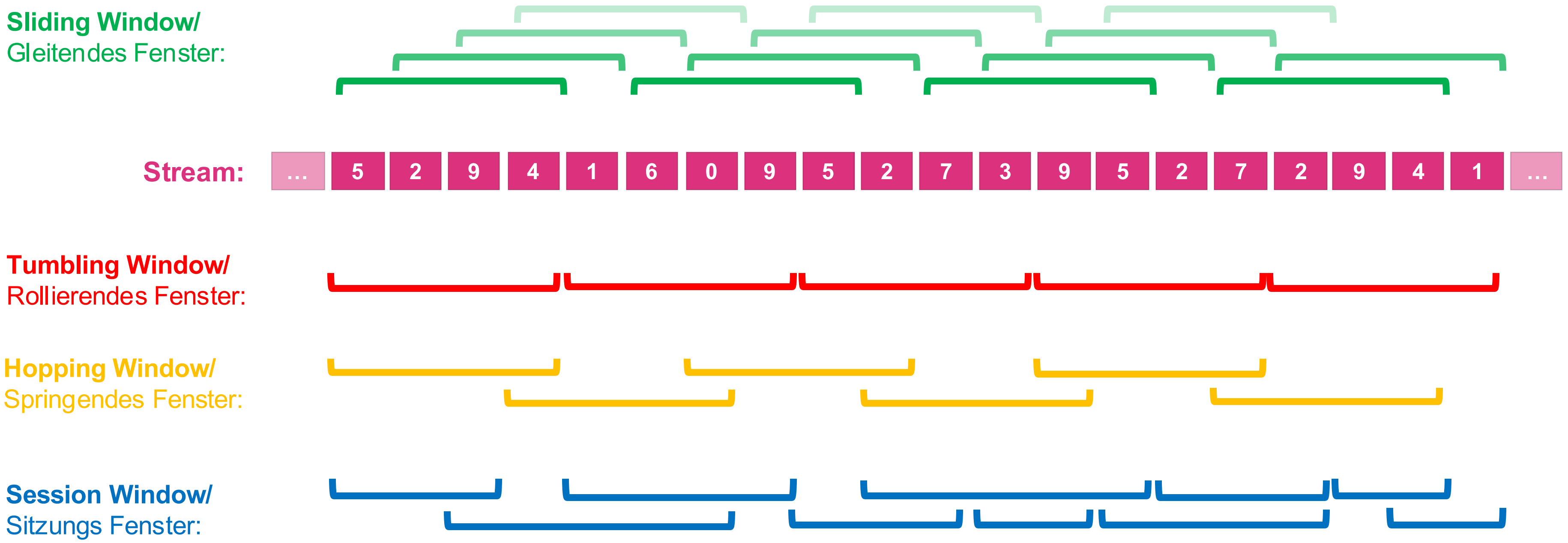
# Message Broker/Queues



# Message Broker/Queues vs Data-Systems

| | Message Broker/Queue | Database |
|---|---|---|
| Persistence: | - **temporarily** (delete once transmitted or after a certain timeframe) | - **persistent** |
| Data Retrieval: | - **Subscription-based** (even if e.g. Kafka supports qu ery-based) | - **Query-based** (execute query to receive data) |
| Communi-cation: | - **Initiated by MB** | - **Initiated by clients** |

# Stream Processing – **Windows**

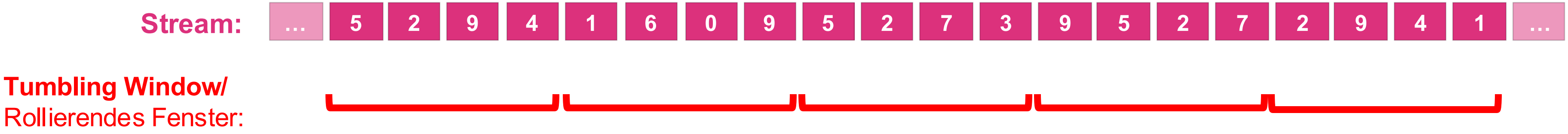


# Stream Processing – **Tumbling Windows**

**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)
```

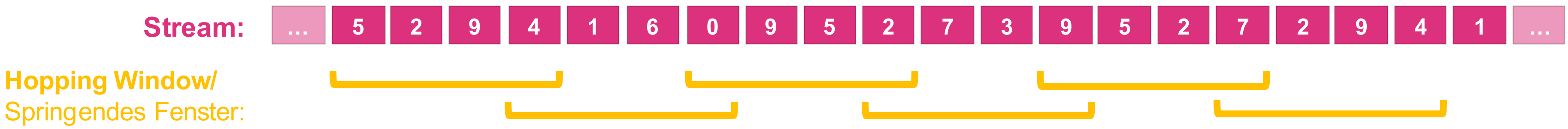


# Stream Processing – Hopping Windows

**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 (dependending on event)
- An event can relate to more than one window
- Event-based, every window contains at least one event and is continuesly slided forward

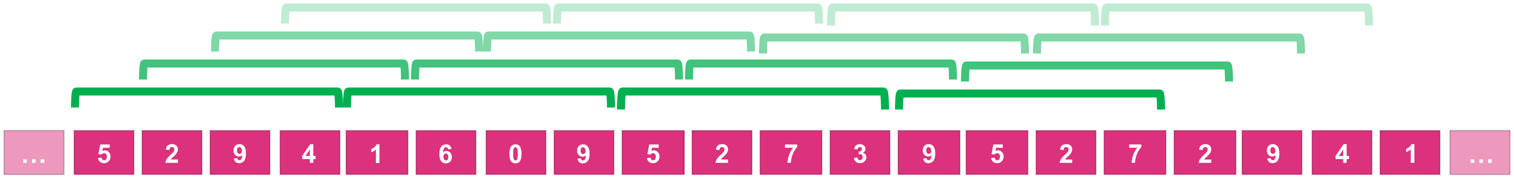
**Twitter Example:**

- 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:**



# 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

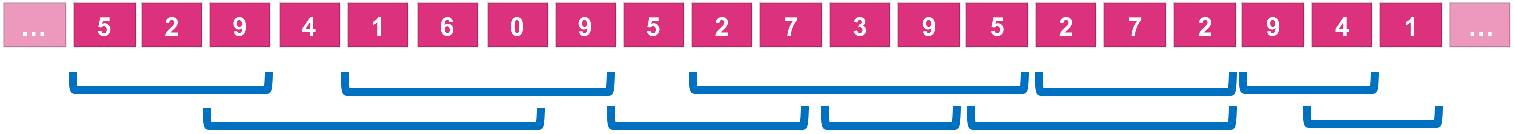
**Twitter Example:**

- 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)
```

**Stream:**

**Session Window/**
Sitzungs Fenster:



# 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, …**

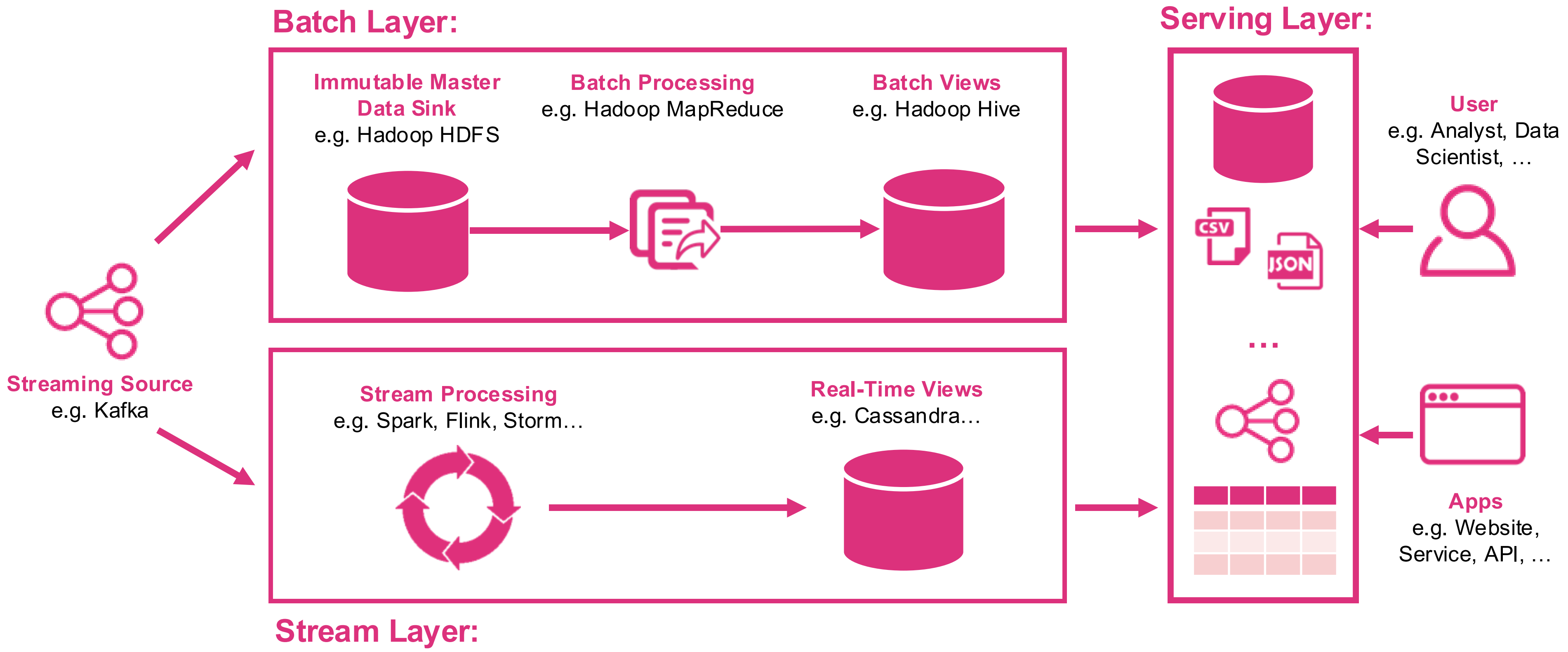
# Word Count on Stream – **Apache Flink**

```
// 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**

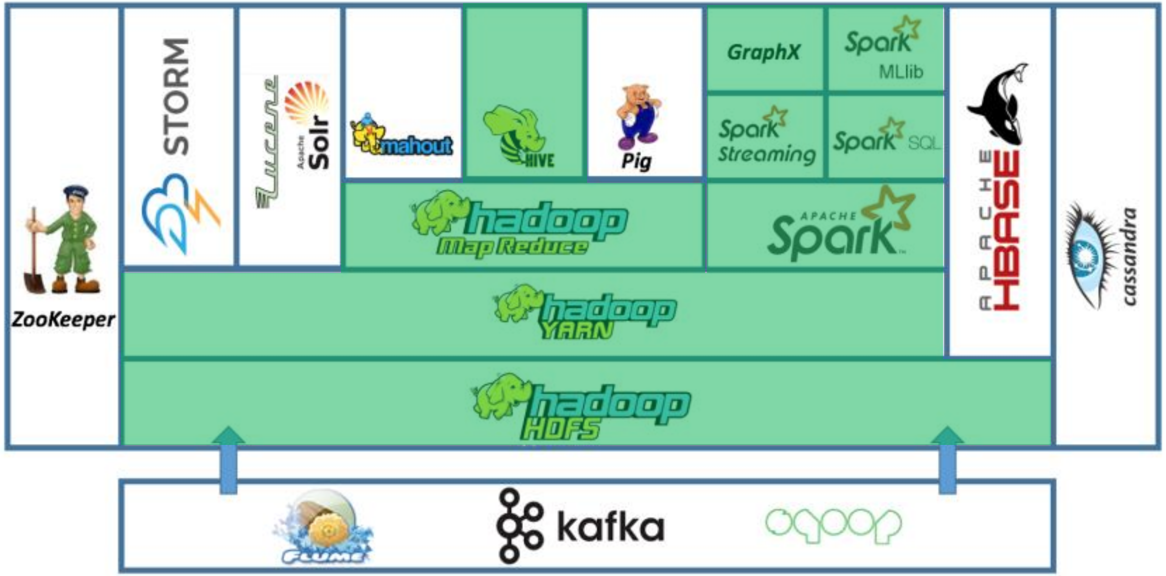
A quick Introduction to ETL Workflow with Pentaho Data Integration

# The Hadoop Ecosystem

Airflow

pentaho

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:

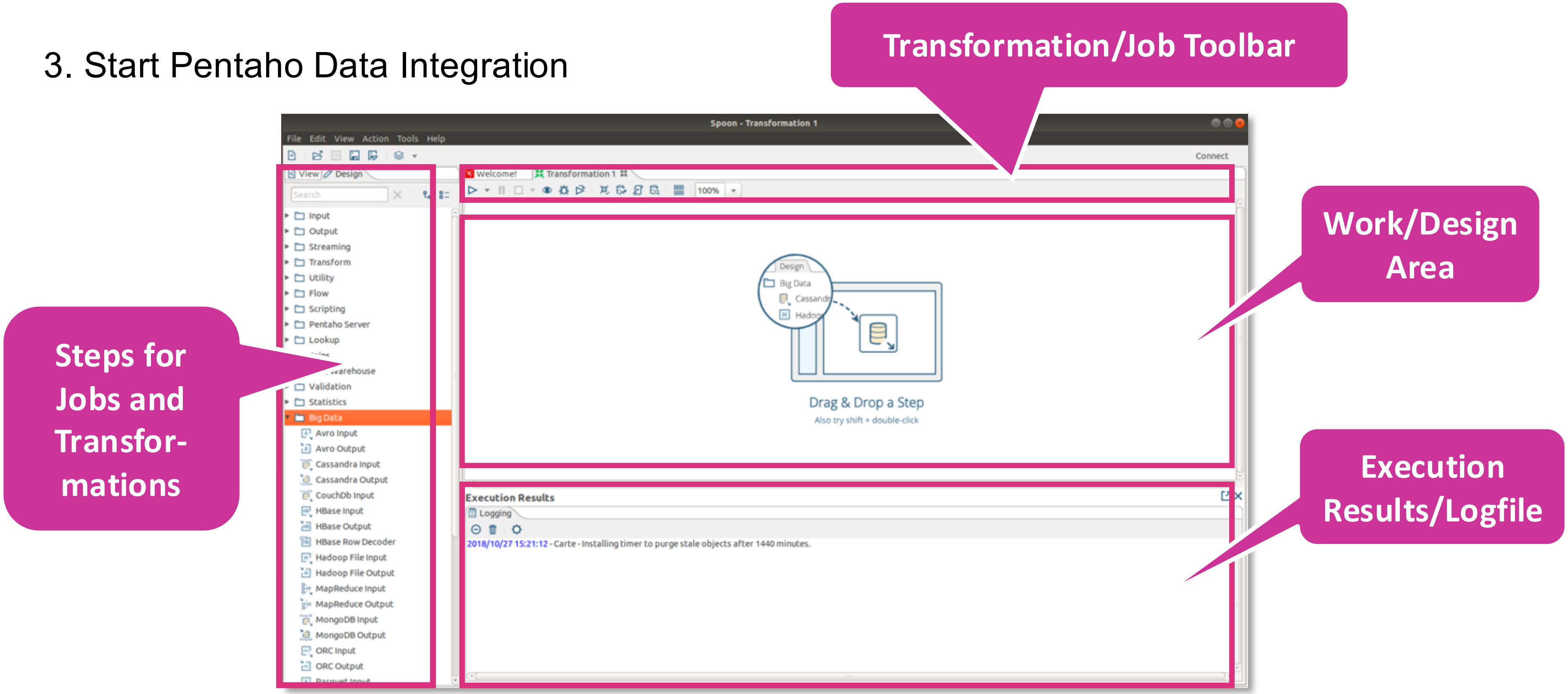
```
/your/pentaho/directory/spoon.sh
```

Windows:

```
/your/pentaho/directory/spoon.bat
```

# Spoon Interface

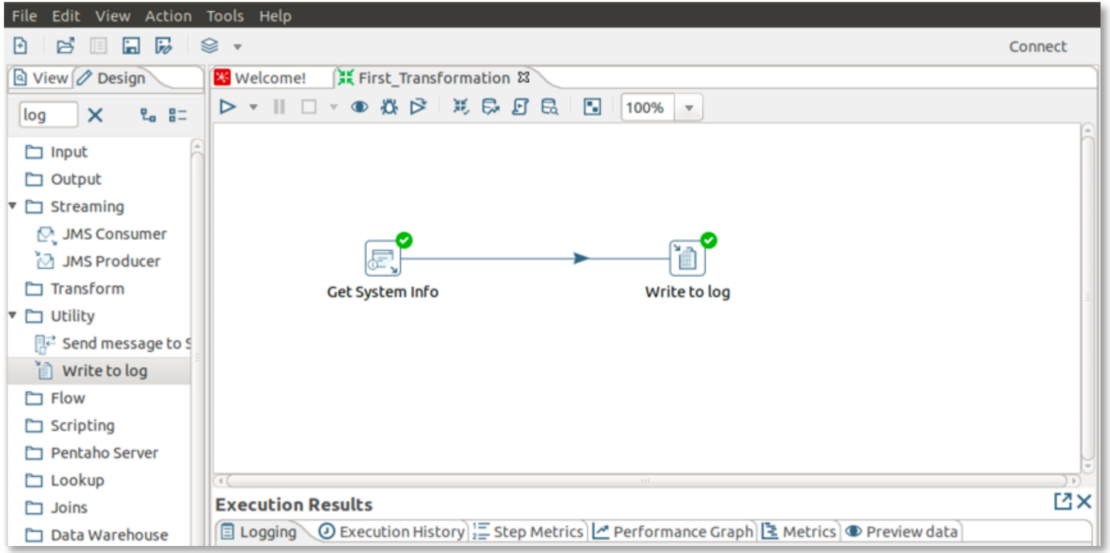
3. Start Pentaho Data Integration



# PDI Basics/Examples – **First Transformation**

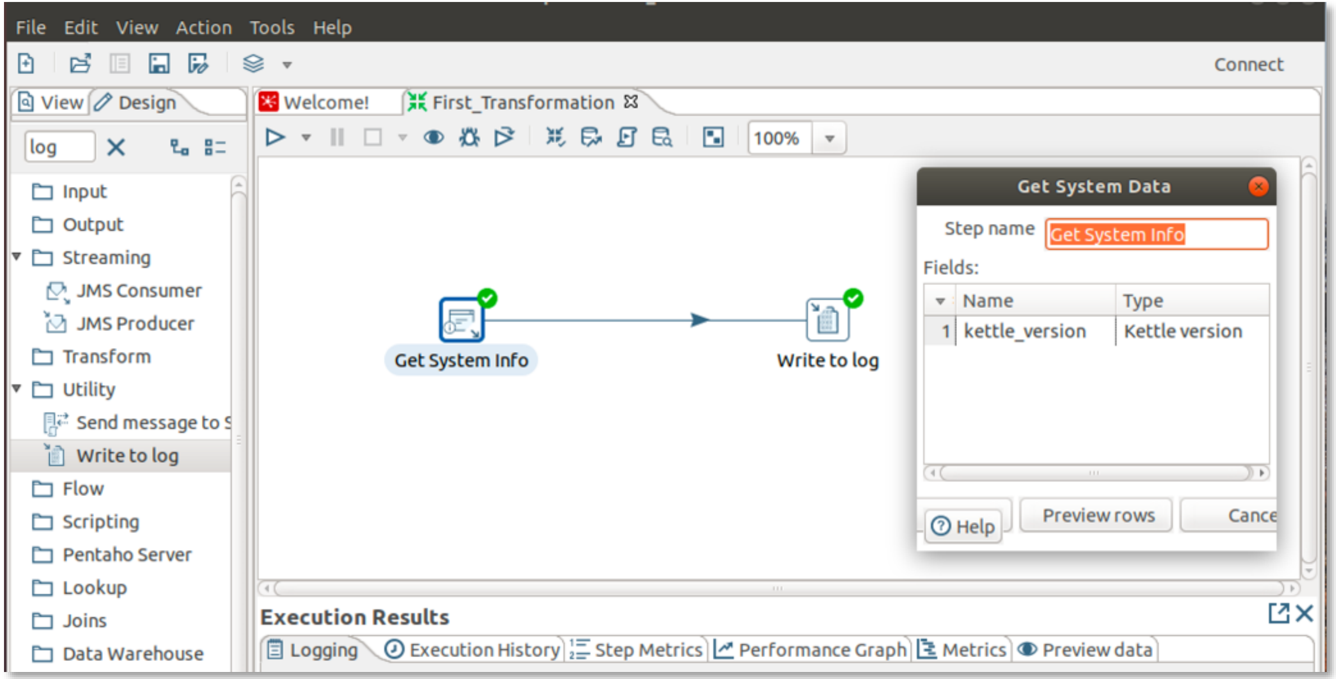
1. Create new **Transformation** (Click: *File → New → Transformation*)

2. Drag&Drop step „*Get System Info*“ and „*Write To Log*“ into Work/Design Area and connect both steps:



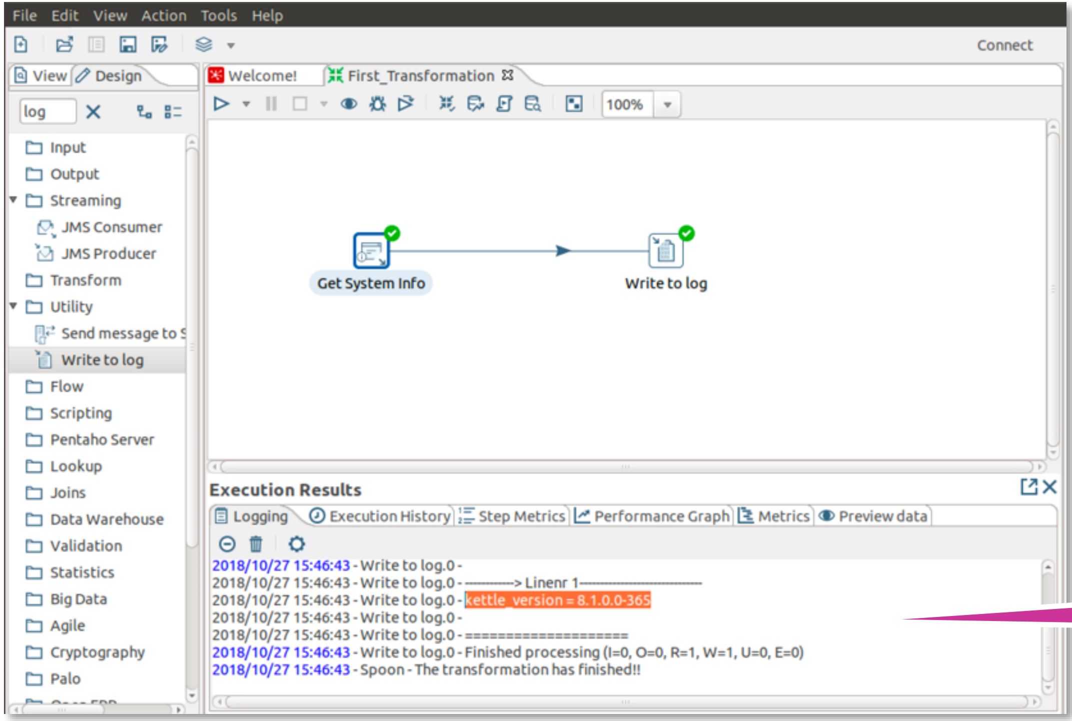
# PDI Basics/Examples – **First Transformation**

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:



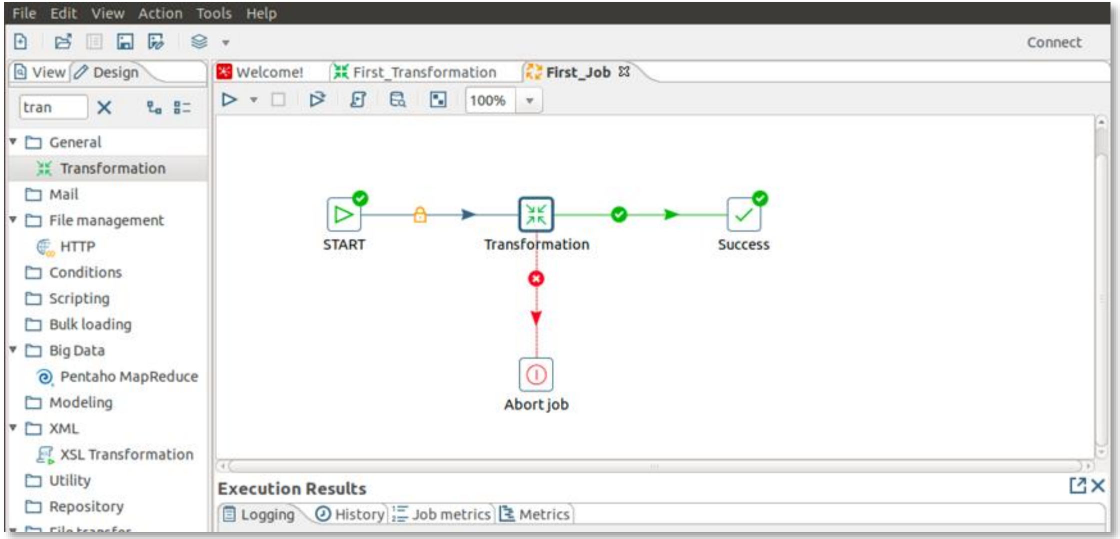

- results of „*Get System Info*“ step are redircected to „Write To Log“ step
- „Write to Log“ step will write results to execution log
- **transformations** are all about actual data transform ation and data flow

**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

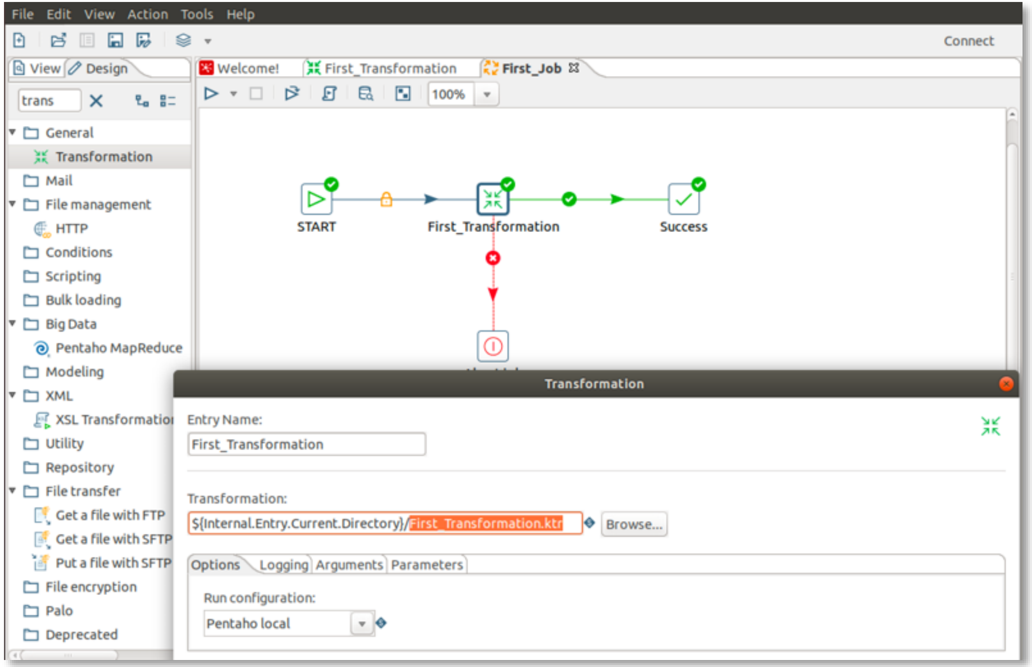

- End of a Job (if not successful)


- End of a Job (if successful)


- Includes a Transformation

# PDI Basics/Examples – **First Job**

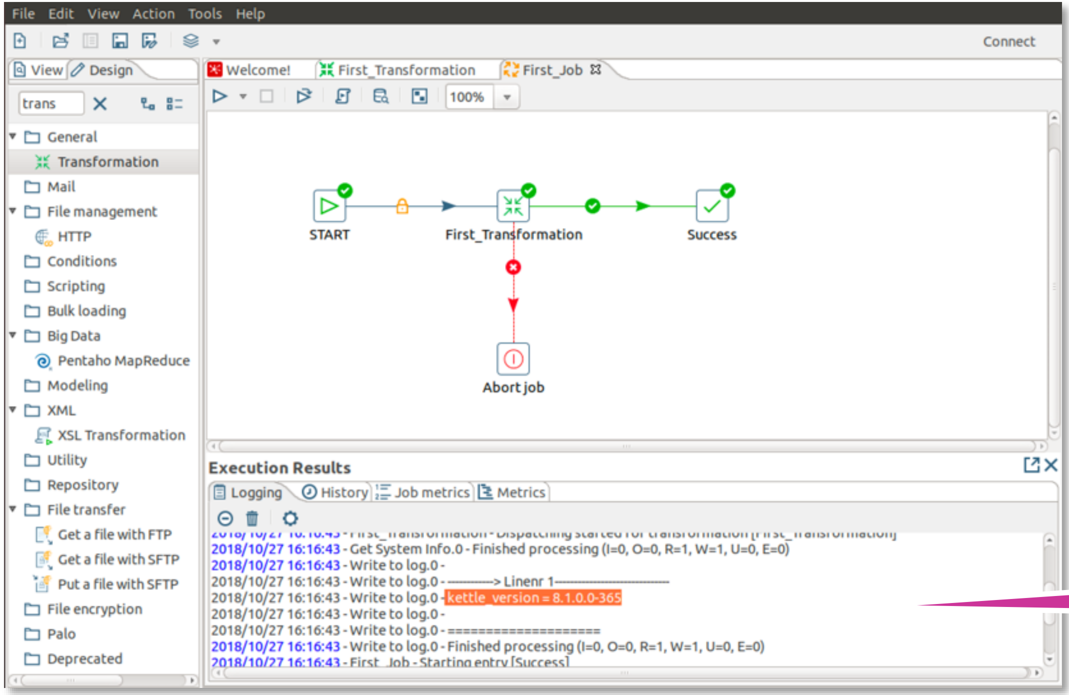
3. Include previously created Transformation (First_Transformation) by double click on step „*Transformation*“ and including „*First_Transformation.ktr*“:




www.marcel-mittelstaedt.com

# 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

# 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)
```

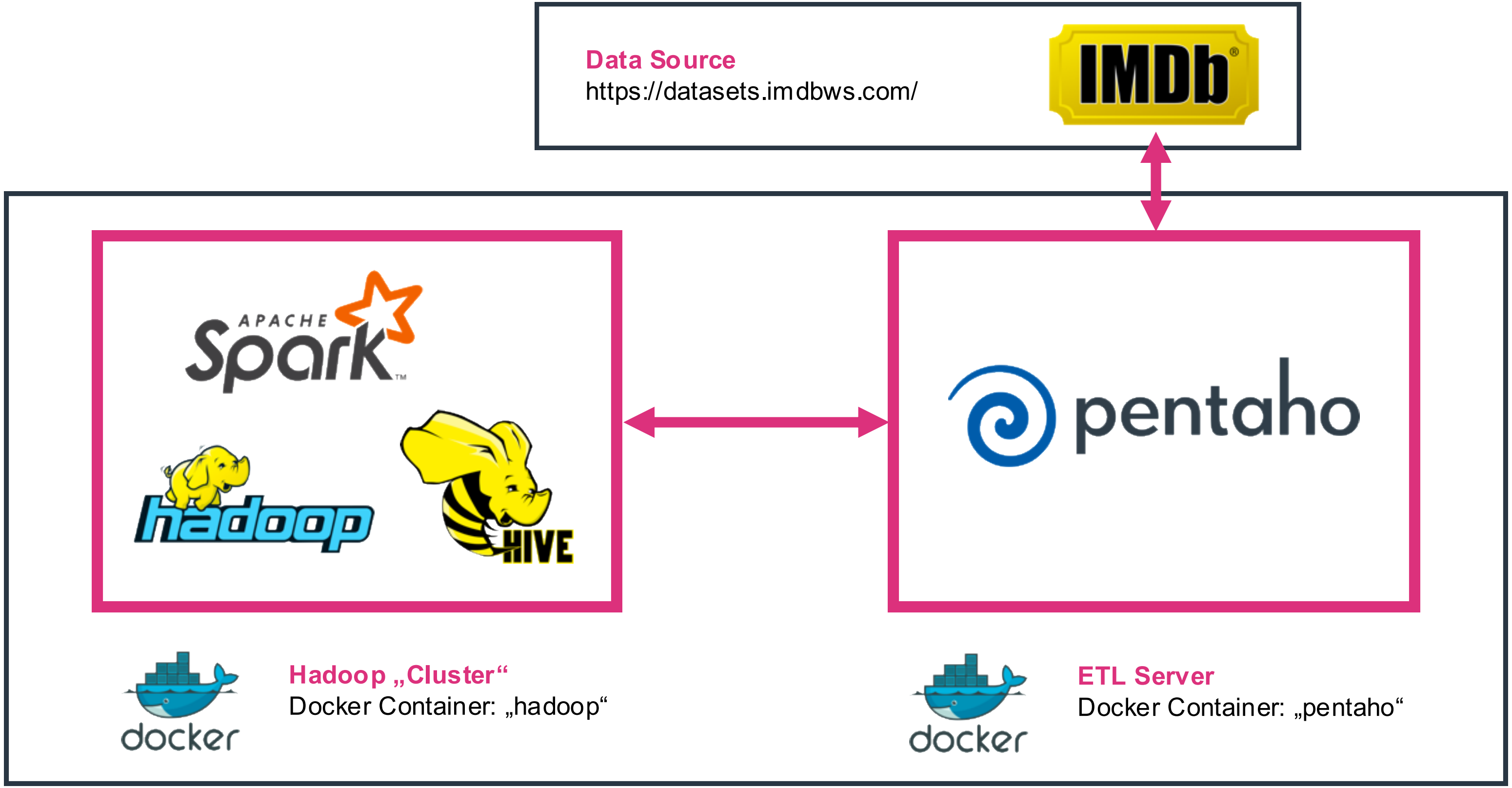
# Break

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?




**Gcloud Server** (Connected to via SSH)

# 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/slf4
j/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 - ketle_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.k
tr]
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 - ketle_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

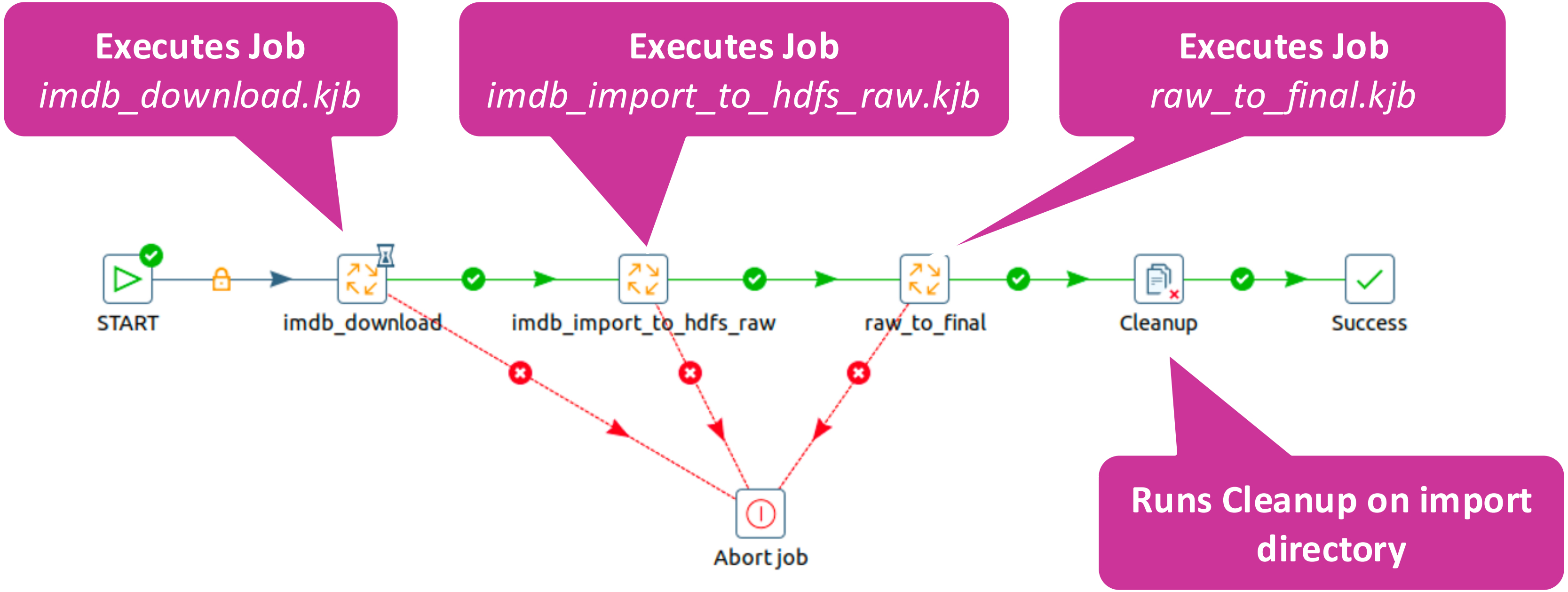
Now let‘s take a look at a real world example: **IMDB data**

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

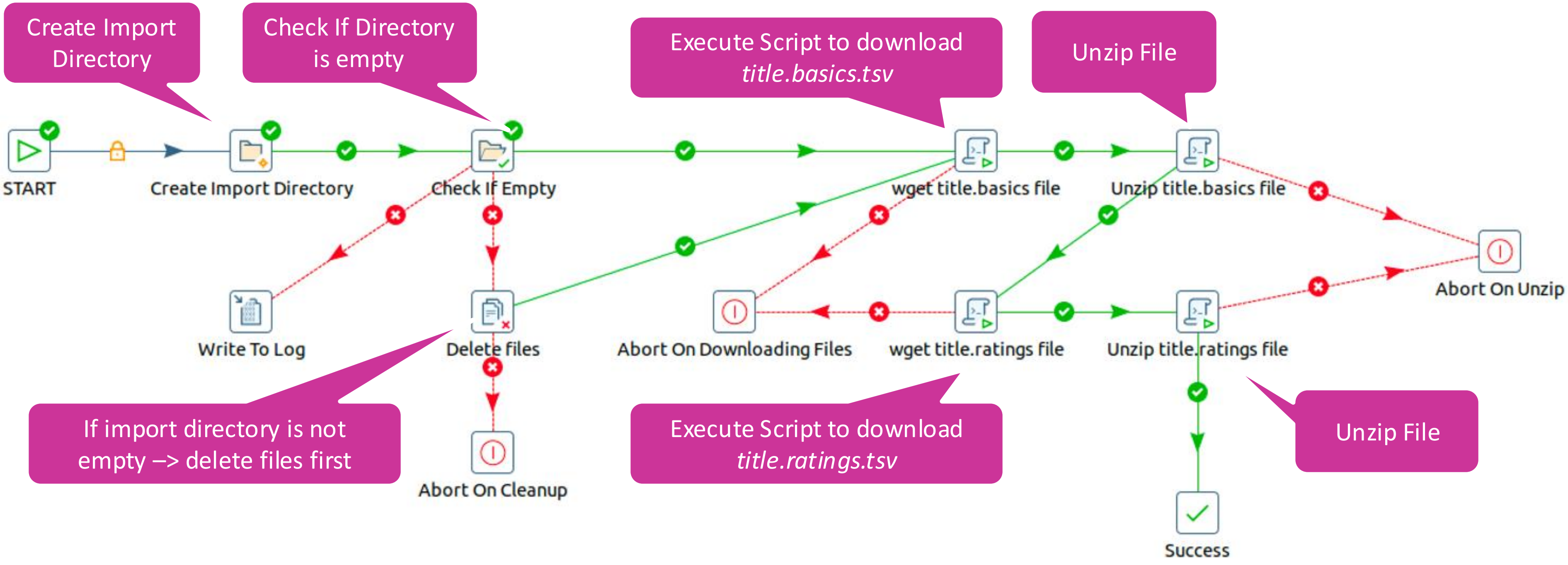
A very simple and naive ETL workflow which is able to run each day:

- **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

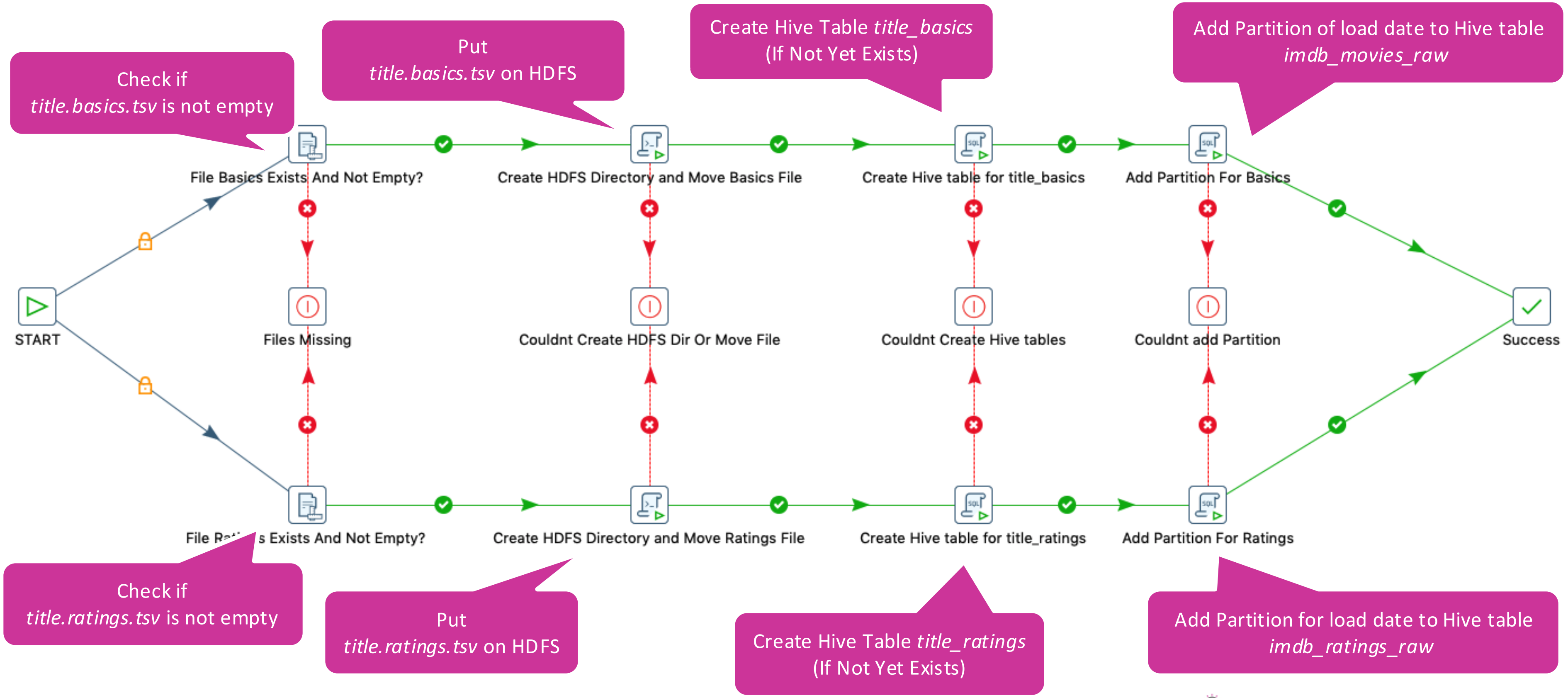
# PDI IMDb Import – **Main Job** *(imdb_main.kjb)*



# PDI IMDb Import – *imdb_download.kjb*



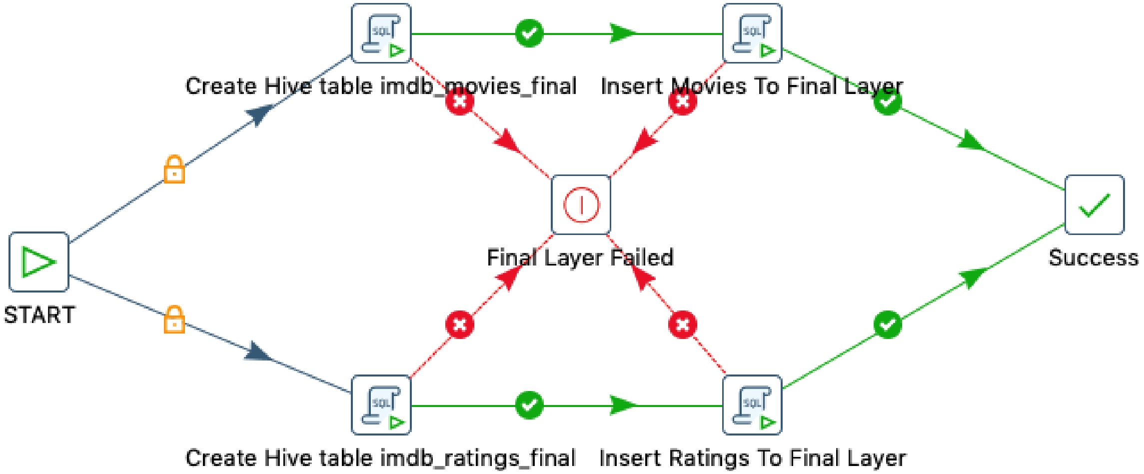
# PDI IMDb Import – *imdb_import_to_hdfs_raw.kjb*



# 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**

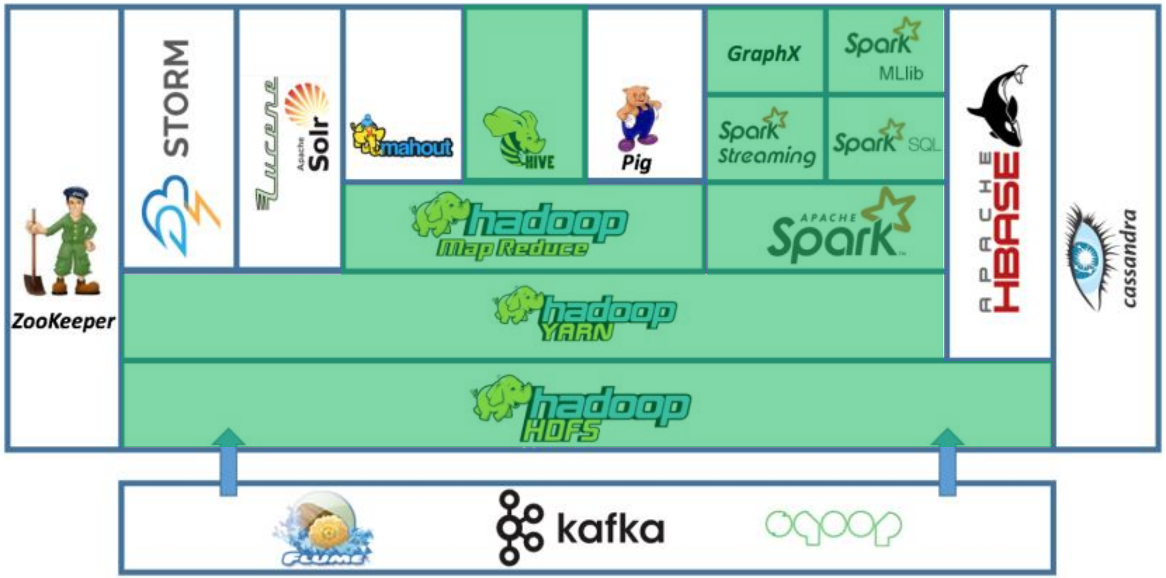
A quick Introduction to ETL Workflow with Apache Airflow

# The Hadoop Ecosystem

pentaho

Airflow

Today's (exercise) focus

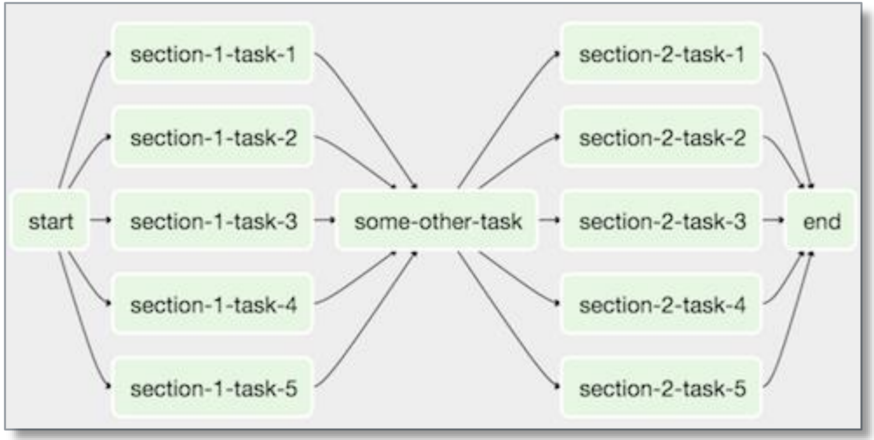


# Airflow

Airflow

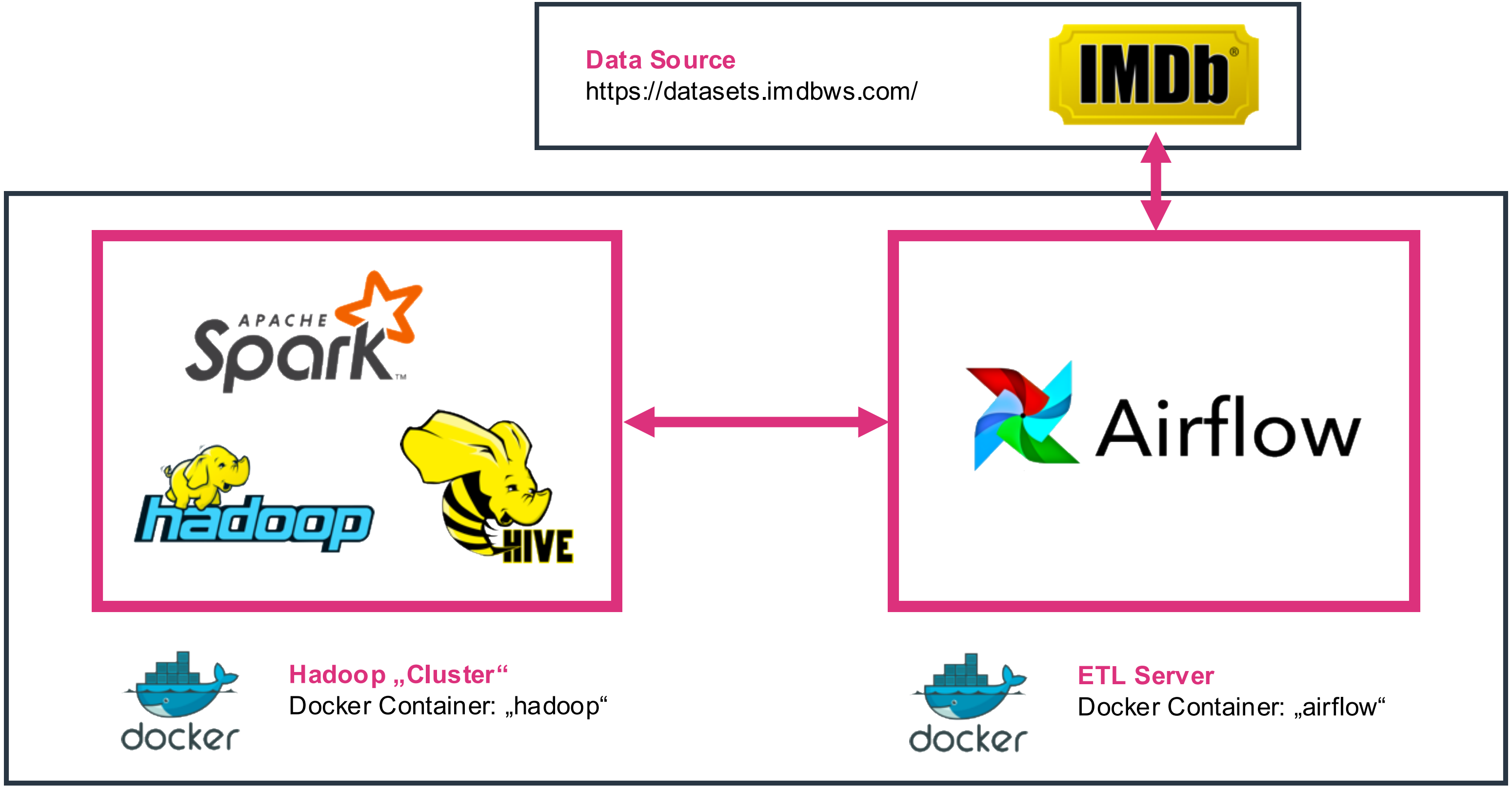
- Apache Open Source Project
- Model Task (e.g. ETL) Workflows
- Python Code Base
- Scheduling, Queues and Pools
- Cluster Ready
- Web UI

- 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?




Gcloud Server (Connected to via SSH)

# 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/slf4
j/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
```

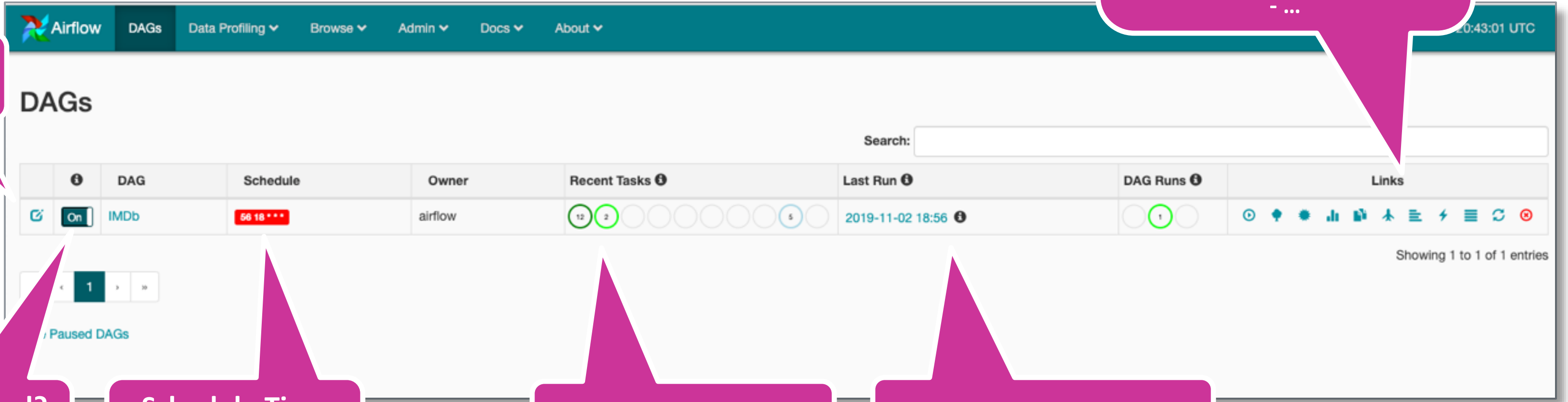
```
cd
```

# Exercises Preparation II

Airflow First Steps/Dag

# 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
- ...

**DAGs**

**DAG scheduled?** (on/off)

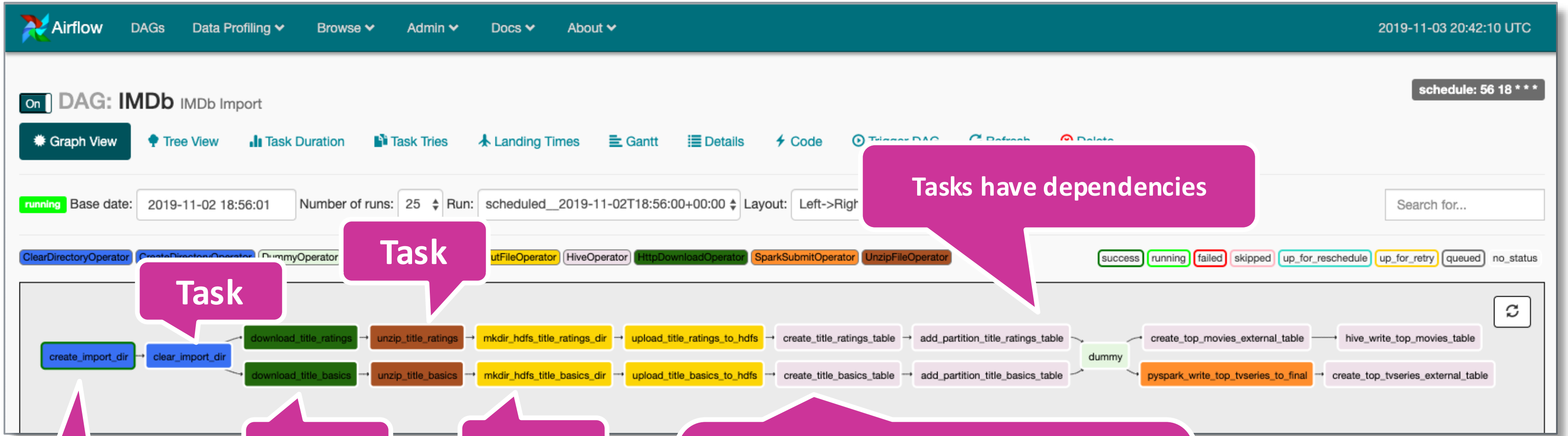
**Schedule Time** (Cron)

**Recent Executions**

**Last Execution of DAG**

# Spoon Interface

Graph View:




Tasks have dependencies

Task

Task

Task

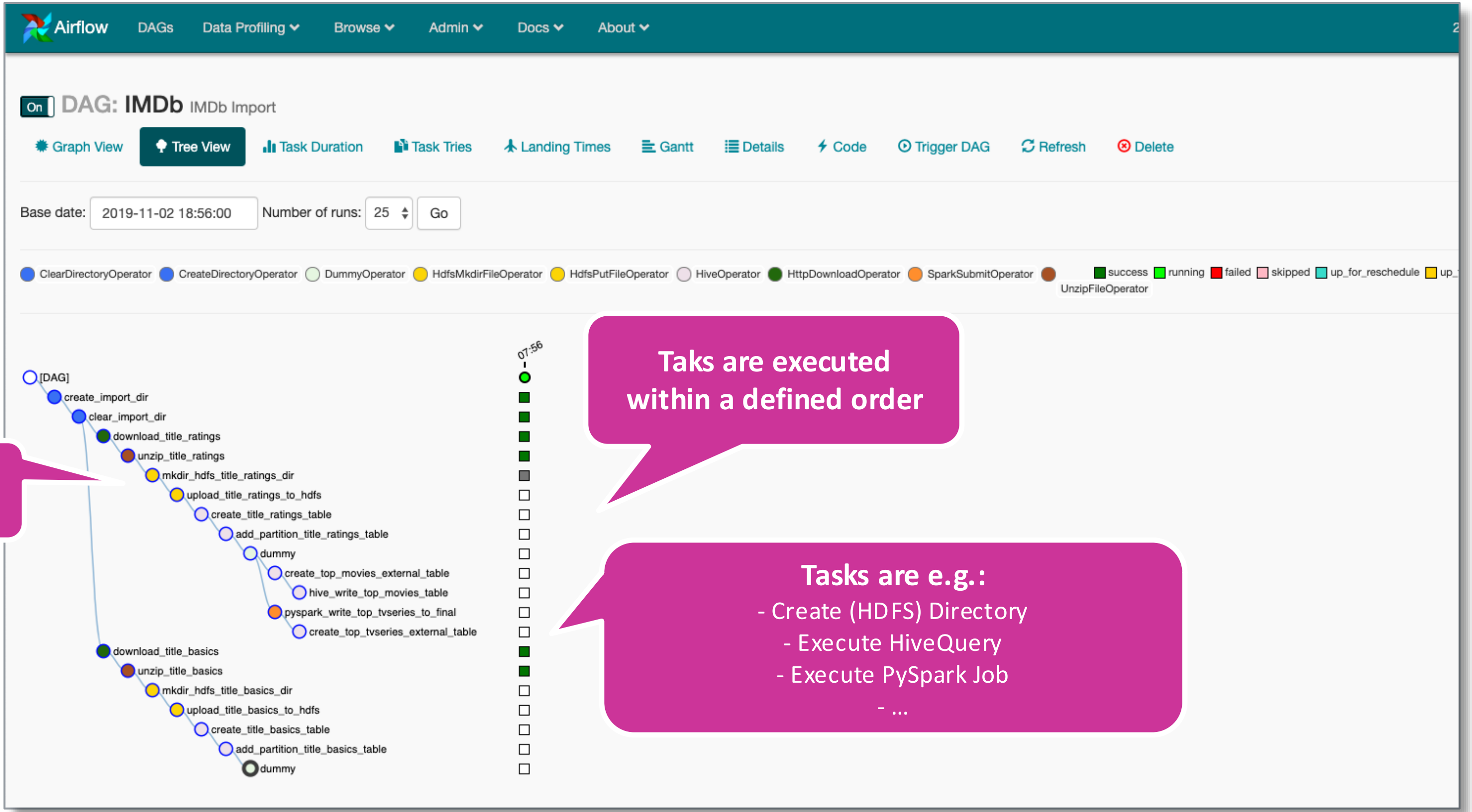
Task

...

**Tasks are e.g.:**
- Create (HDFS) Directory
- Execute HiveQuery
- Execute PySpark Job
- ...

# Spoon Interface

Tree View:



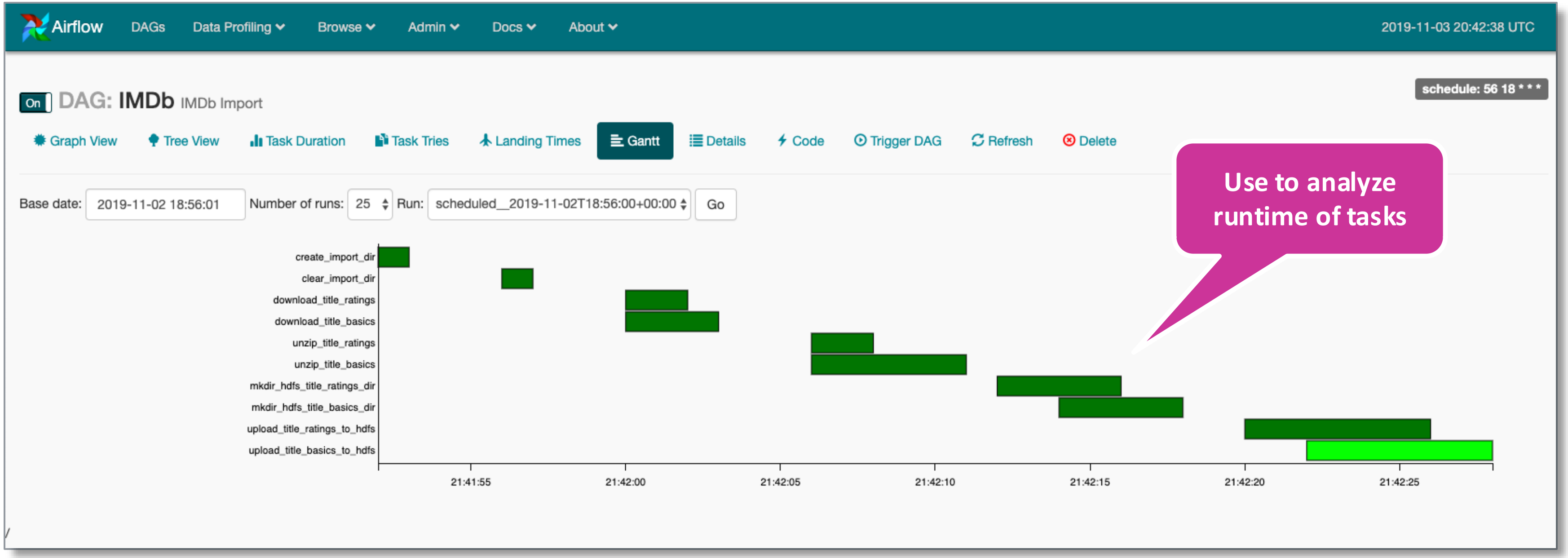

Tasks have dependencies

Taks are executed within a defined order

Tasks are e.g.:
- Create (HDFS) Directory
- Execute HiveQuery
- Execute PySpark Job
- ...

# Spoon Interface

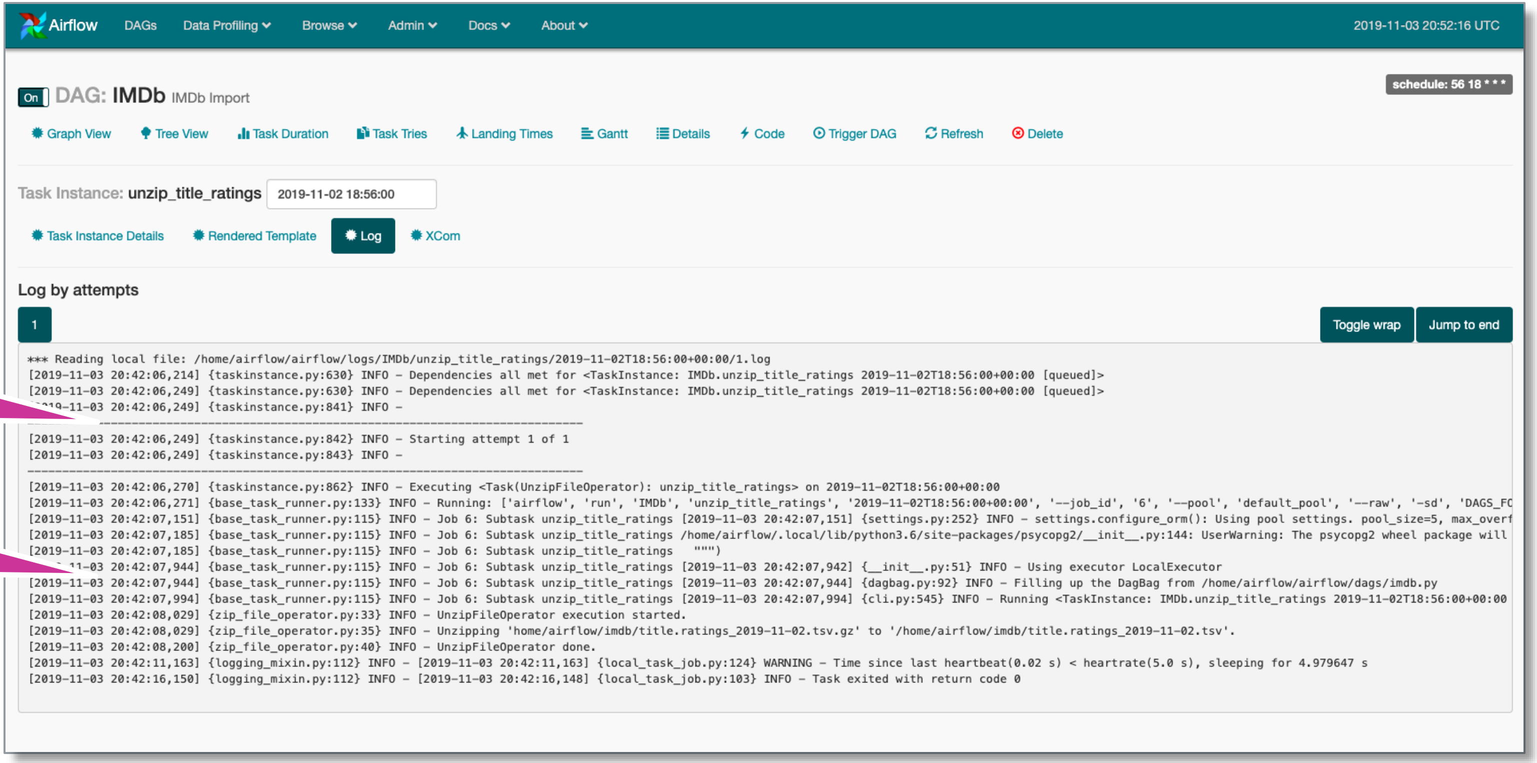
Gantt View:




Use to analyze runtime of tasks

# Spoon Interface

Logs View:




**Use for Debugging**

**Each Task has it's own Log**

# 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
```

**DAG Definition, e.g.**
- Name
- Schedule Interval (Cron)
- Description
- Start date
- …

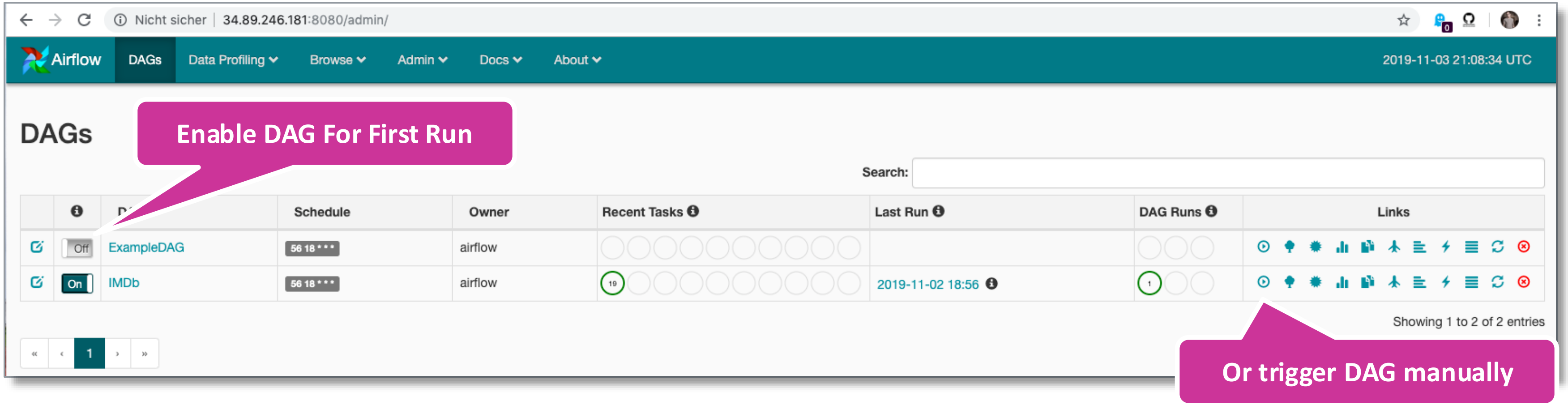
**Task 1**

**Task 2**

**Task Execution Order**
***task_2*** is dependent on **task_1**

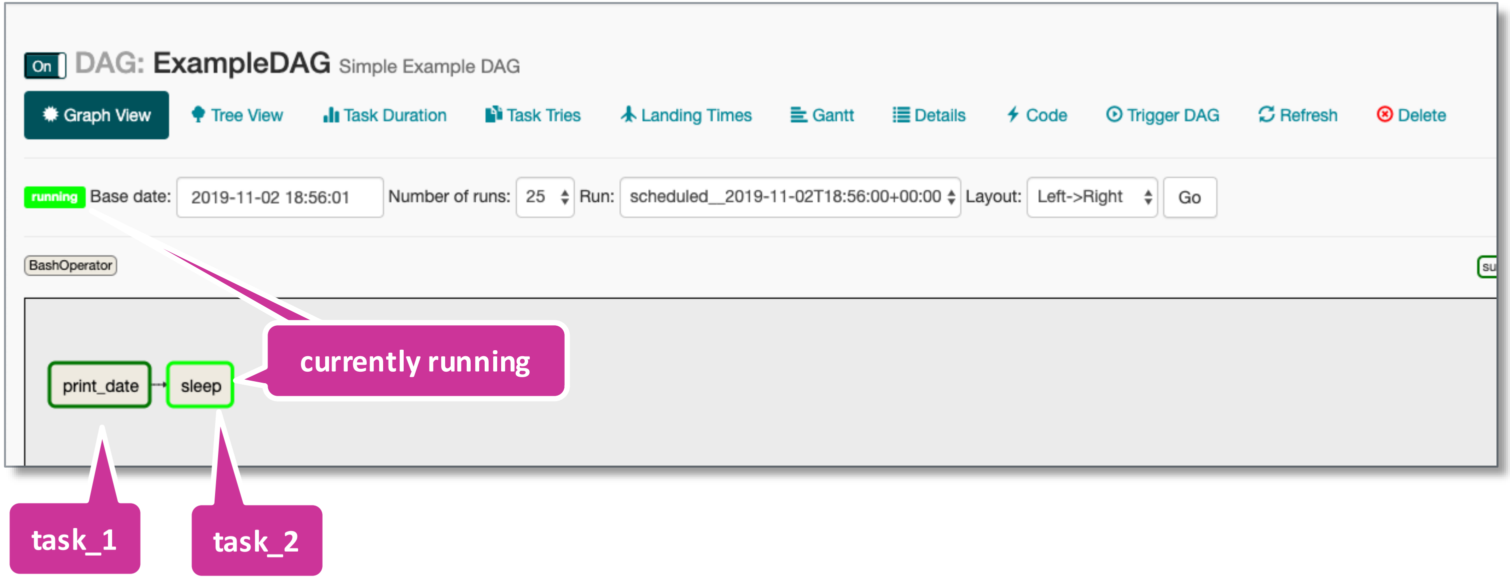
# Spoon Interface

Execute DAG:



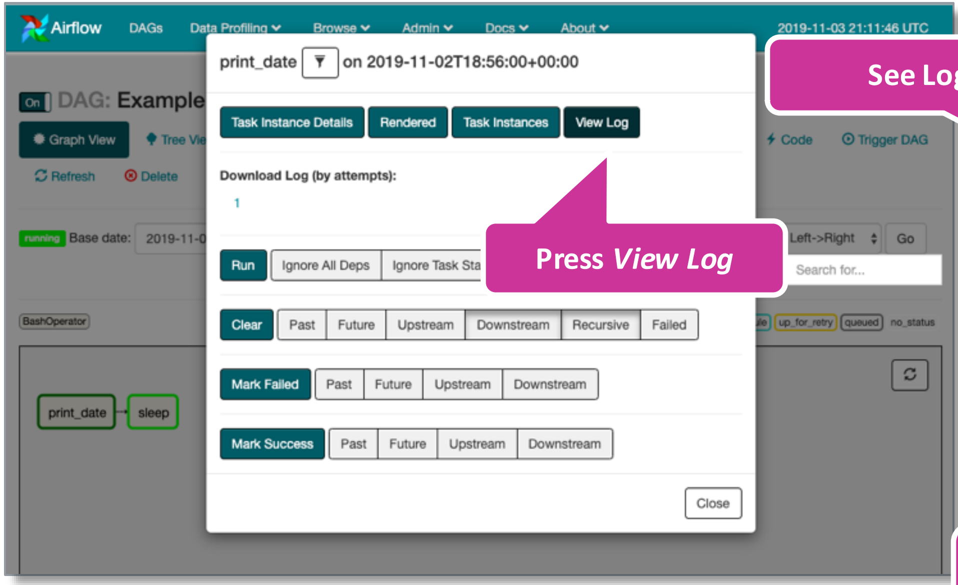
# Spoon Interface

See executing DAG:

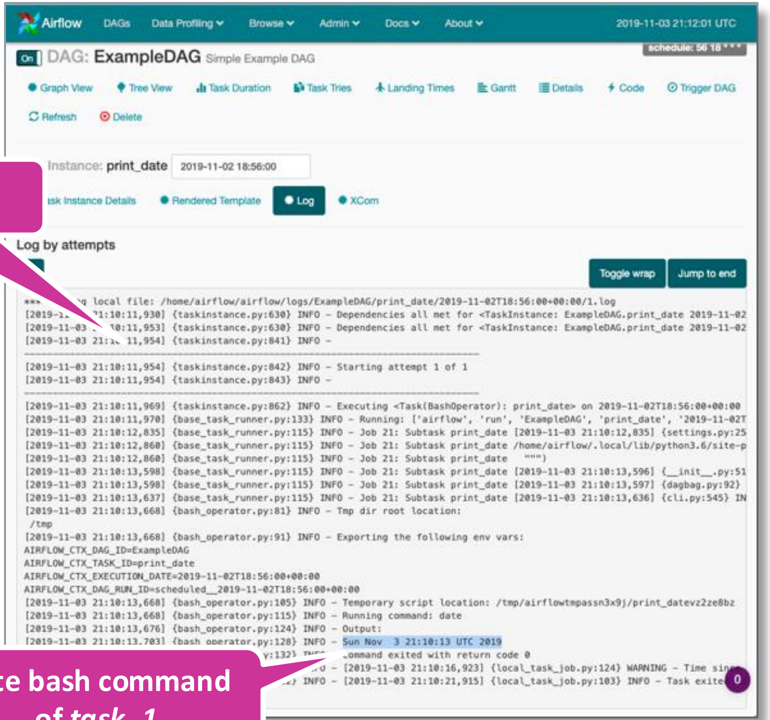


# Spoon Interface

View Log of task *print_date*:




Press *View Log*




See Log

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

# Stop Your VM Instances


```
gcloud compute instances stop big-data
```