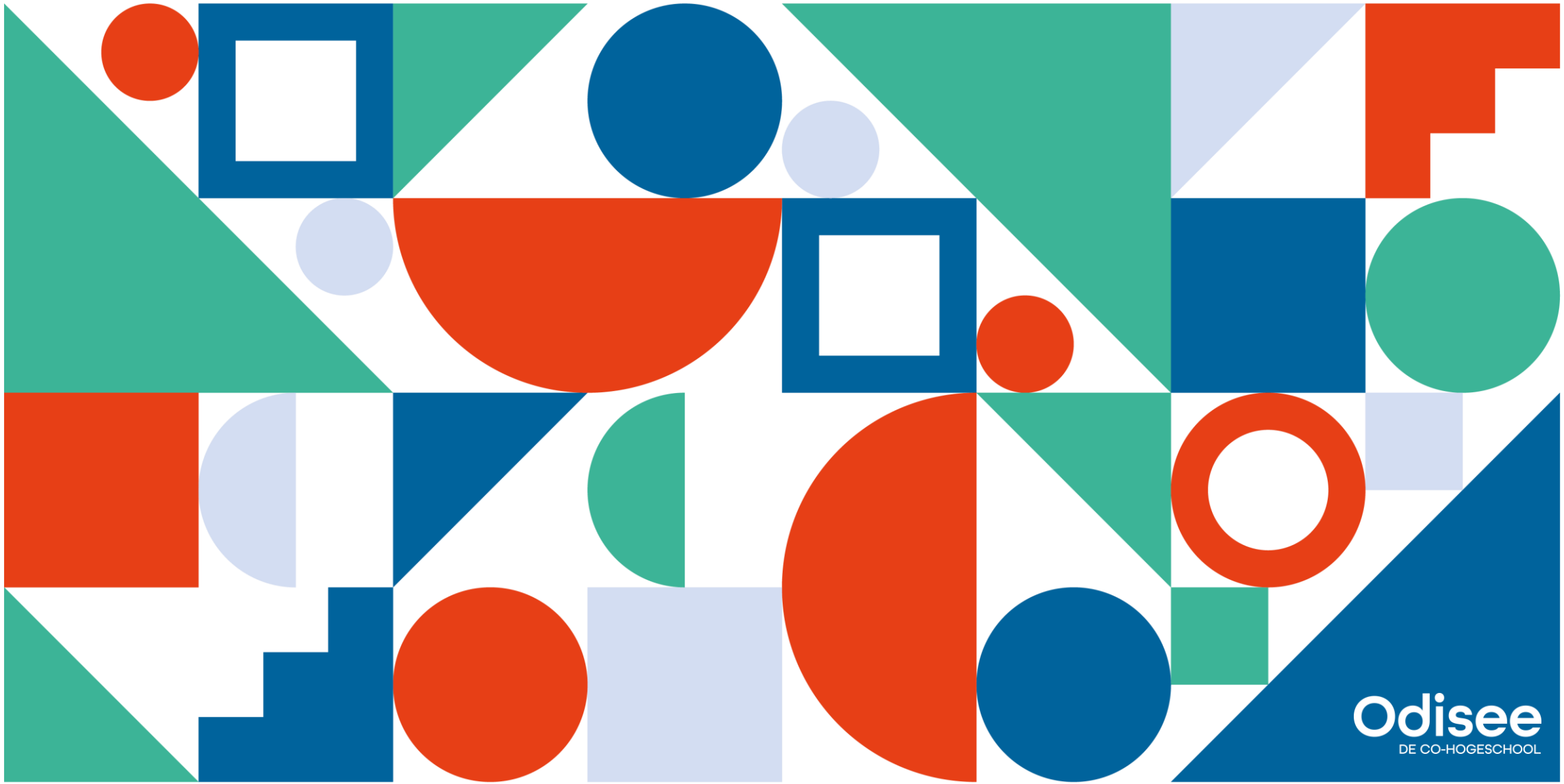
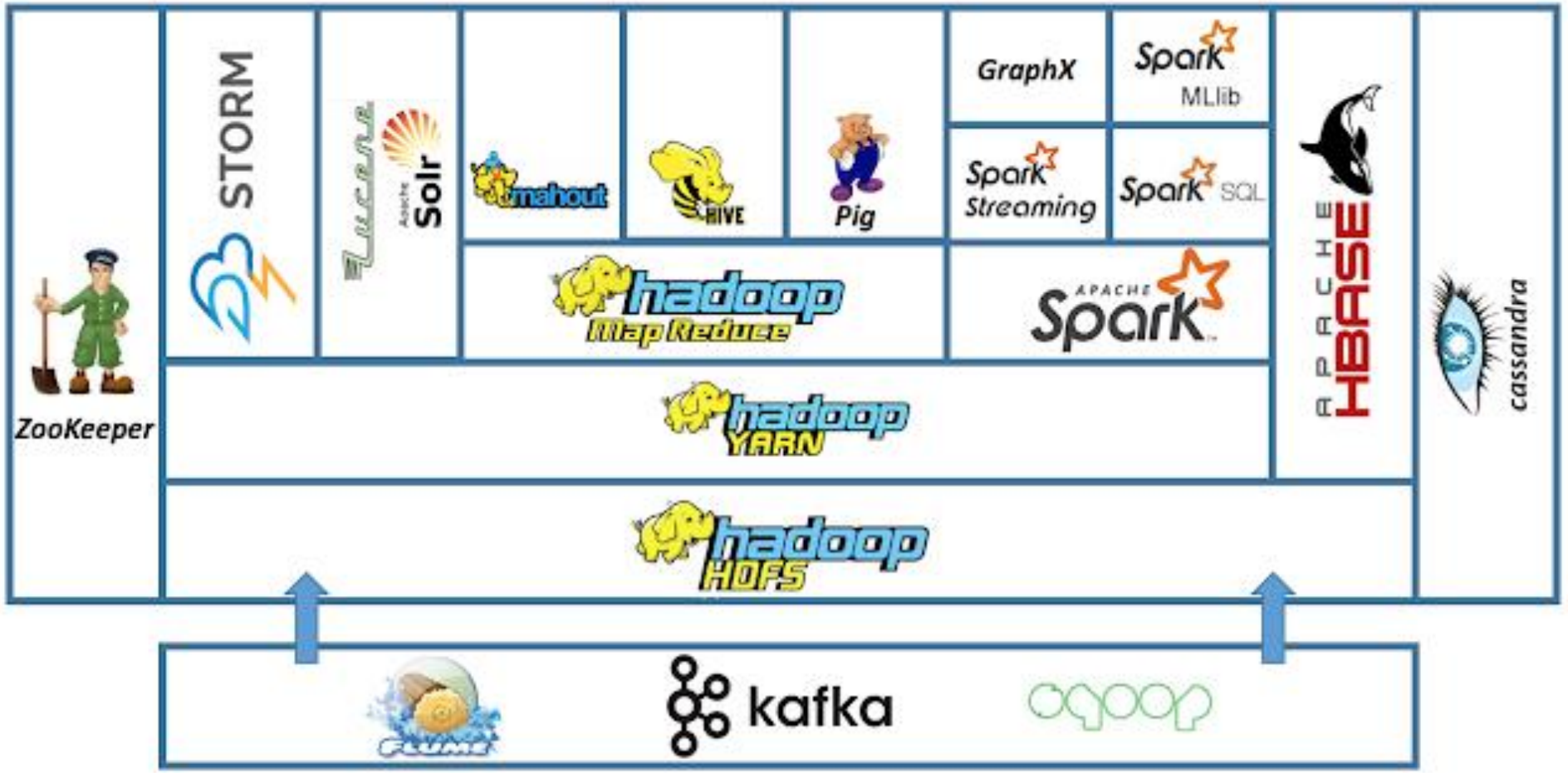

Odisee
DE CO-HOGESCHOOL

Streaming

Jens Baetens


ZooKeeper
STORM
Lucene
Solr
mahout
HIVE
Pig
GraphX
Spark MLlib
Spark Streaming
Spark SQL
hadoop Map Reduce
APACHE Spark
APACHE HBASE
cassandra
hadoop YARN
hadoop HDFS
FLUME
kafka
sqoop

Wat is Streaming?

Streaming

- Framework om continue datastromen te verzamelen, verwerken, filteren, op te slaan, …

- Een aantal vaak voorkomende platforms hiervoor:
  - Spark Streaming
  - Storm
  - Kafka
  - …

Kenmerken

- Stukjes van data per keer verwerkt
- De eigenschappen van de stream/data kunnen wijzigen
- Volledige dataset moet niet tegelijkertijd beschikbaar zijn

- Past goed binnen het concept van data-lake
  - Lake bevat alle data
  - Streams monitoren folders / spouts om nieuwe data op te vangen en te verwerken
    - ETL paradigma

https://youtu.be/ds7tx5IF8GM?t=22

Spark Streaming

- API sterk gelijkaardig aan Spark
- Maakt gebruik van DataFrames
- Werkt samen met andere Spark-delen zoals MLlib

- Kan op Hadoop cluster runnen

Storm

Storm

- Gratis en open-source
- Distributed real-time computing system
  - Unbounded streams of data
  - Doet wat Hadoop doet voor batch processing
- Gelijkaardig aan Spark Streaming

- Werkt met elke programmeertaal
  - eventueel wel kleine, zelf te schrijven wrapper nodig

Use cases

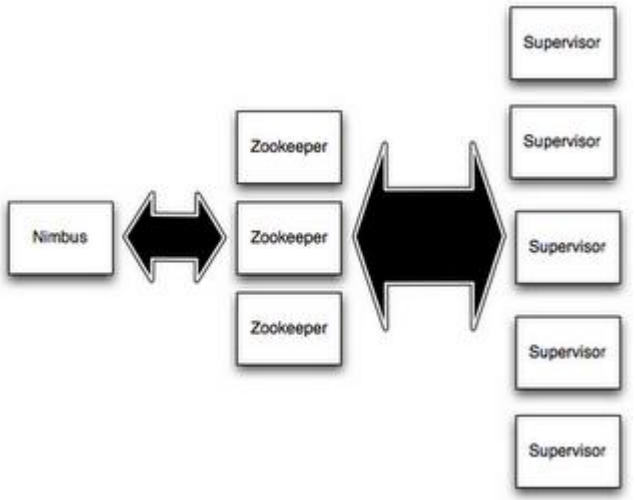
- Realtime analyses
- Online Machine Learning
- Extract-Transfer-Load

Kenmerken

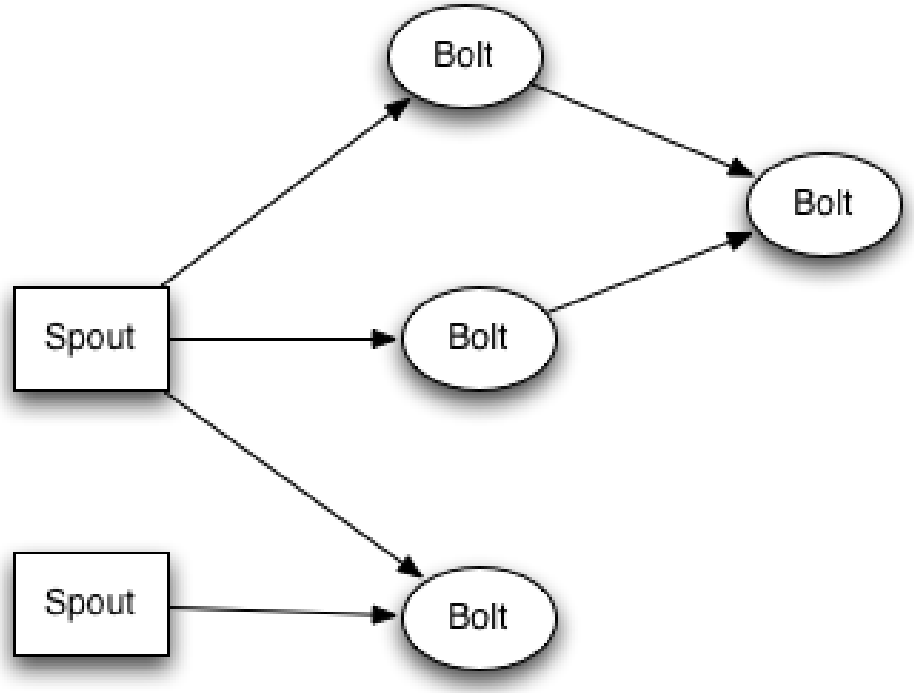
- Heel snel
  - 1.000.000+ tuples / sec / node
- Schaalbaar
- Fout-tolerant
- Garantie dat de data verwerkt wordt
- Gebruiksvriendelijk
- Integreert met bestaande queueing en database technieken

Storm cluster

- Nimbus
  - Master node
  - job tracker binnen Hadoop
- Supervisors
  - Worker nodes
  - Deel van de bewerkingen
- Zookeeper
  - Extra cluster voor beheer en coordinatie



Topologies



- Graaf met de uit te voeren stappen
  - Knopen zijn de bewerkingen
  - Verbindingen de data die gestuurd wordt
- Twee types nodes
  - Spouts / kranen / databronnen
  - Bolt: verwerkt inputs en stuurt eventueel een nieuwe stream uit

Kafka

Capaciteiten

- Aanmaken van streams (write)
- Streams binnenhalen (read)
- Streams opslaan
- Streams verwerken

- Highly scalable, secure, fault-tolerant

Componenten

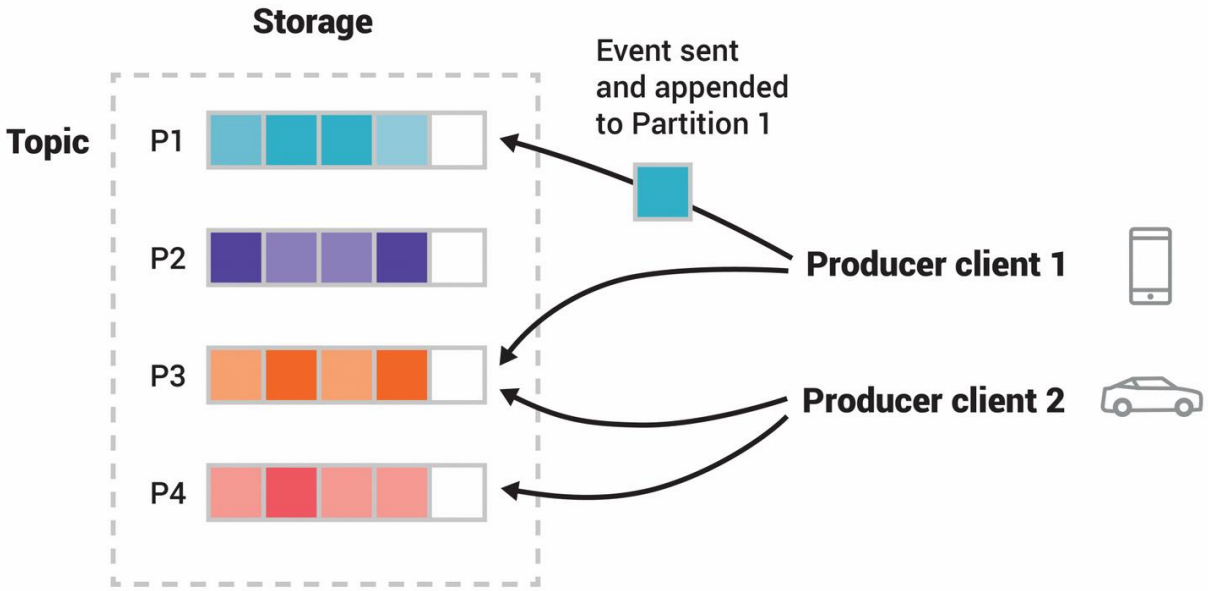
- Servers
  - Cluster dat meerdere datacenters/regios kan omvatten
  - Types
    - Storage layer -> brokers
    - Kafka connect -> import en export data als streams
- Clients
  - Voer gedistribueerde applicaties en microservices uit om de streams te verwerken

Concepts / terminology

- Event (record/message)
  - Doorgestuurd in de streams
  - Key, value, timestamp, metadata (optional)
- Producers: Write
- Consumers: Read and process
- Topics
  - Alle binnengekomen events (niet verwijderd na gelezen/remain time instelbaar)
  - Meerdere consumers / producers

Topics

- Kunnen verdeeld worden over een aantal buckets op verschillende brokers
- Belangrijk voor schaalbaarheid
- Zelfde event-key = zelfde partitie
- Kan gerepliceerd worden



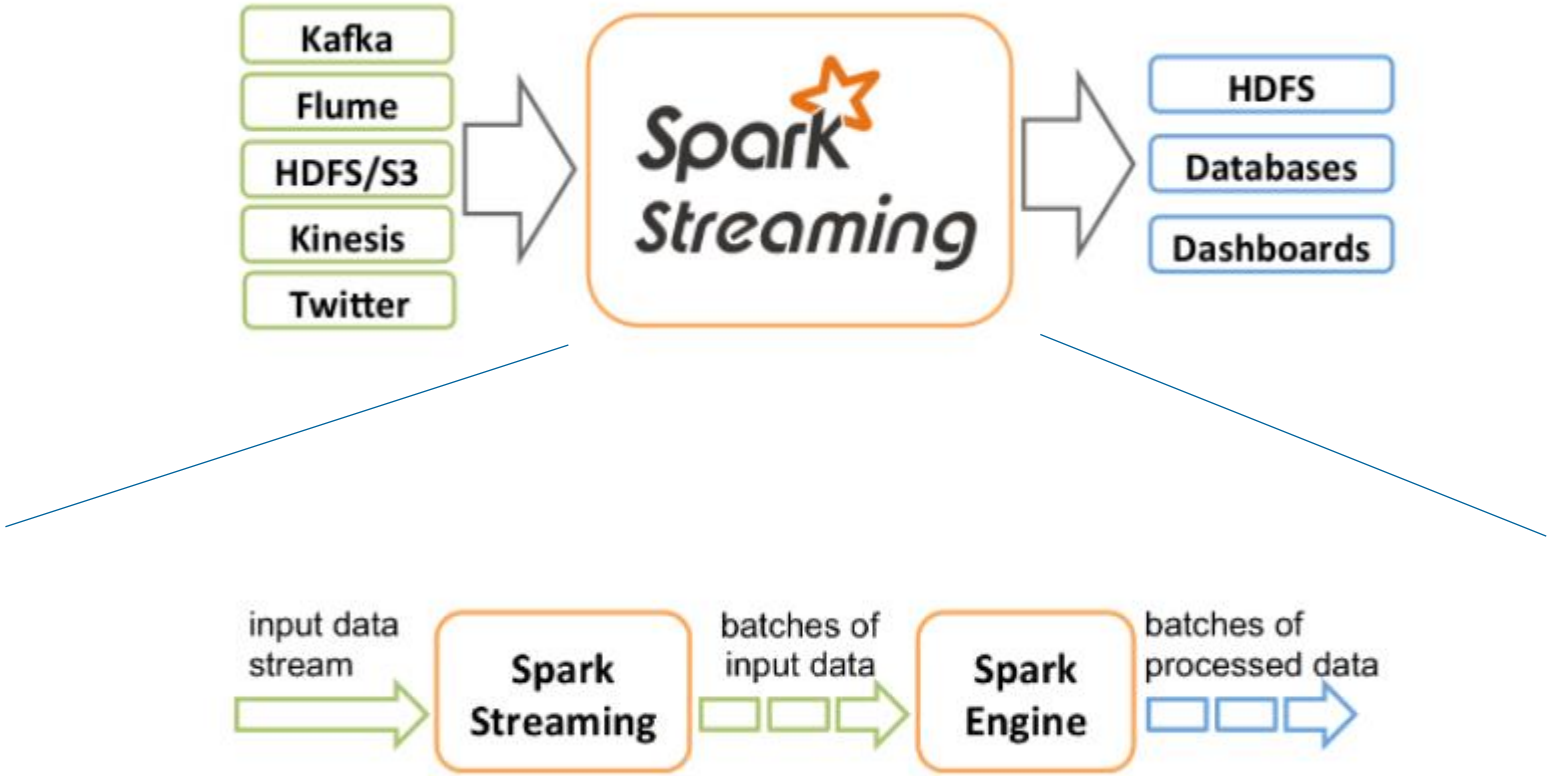
https://youtu.be/Ch5VhJzaoaI

Wanneer wat?

- Apache Spark
  - Schaalbare, fout-tolerante streaming applicaties
- Apache Storm
  - Distributed real-time berekeningen
- Apache Samza
  - Real-time data verwerking met state
- Apache Flink
  - Data Stream berekeningen met state
- Amazon Kinesis Data Streams
  - Real-time managed data streaming
- Apache Ignite
  - High performance in-memory computing

Spark Streaming

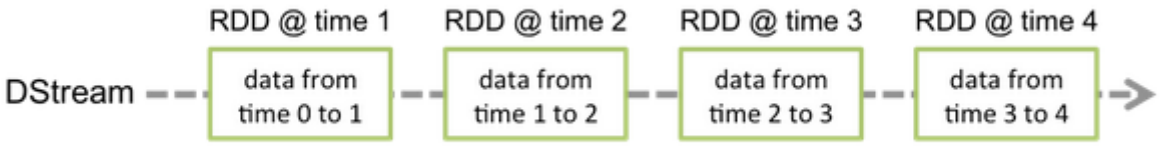
Inlezen


Kafka
Flume
HDFS/S3
Kinesis
Twitter
Spark Streaming
HDFS
Databases
Dashboards
input data stream
Spark Streaming
batches of input data
Spark Engine
batches of processed data

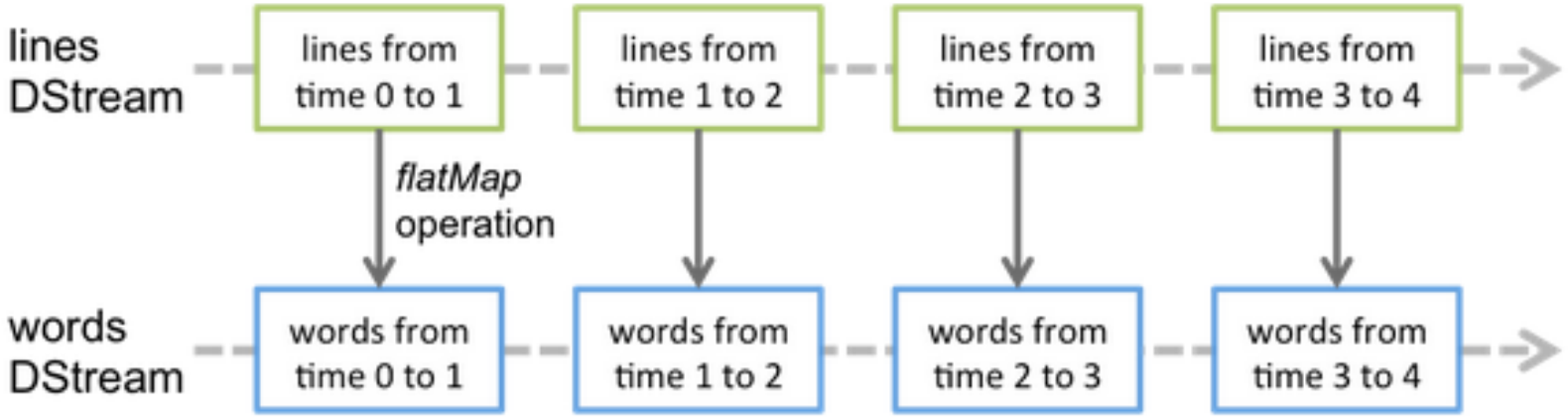
https://youtu.be/dYBWZTZT6o0

DStreams

- Basisabstractie van SparkStreaming
  - Continue datastroom
    - Ontvangen van input
    - Verwerkte datastroom (load)
  - Reeks van RDD's
    - Elke RDD bevat de inputdata ontvangen tijdens een bepaald interval



Streaming – wordcount example



Streamingcontext ipv Sparkcontext

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

# Create a local StreamingContext with two working thread and batch interval of 1 second
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 1)
```

Opmerkingen

- Wanneer spark lokaal uitgevoerd wordt zijn minstens twee threads nodig
- Elke Dstream is verbonden met een receiver
  - Haalt data binnen en slaat het op in het Spark toegewezen geheugen
  - Aantal cores toegekend aan spark moet minstens 1 hoger zijn dat het aantal receivers

Sources

- TCP-socket

```
# Create a DStream that will connect to hostname:port, like localhost:9999
lines = ssc.socketTextStream("localhost", 9999)
```

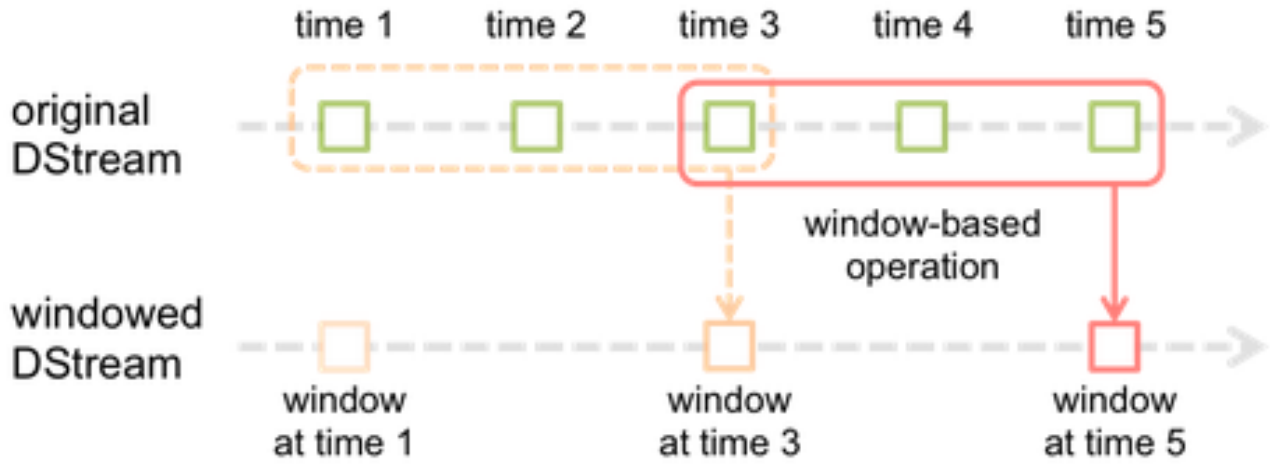
- Monitor Directory

```
streamingContext.textFileStream(dataDirectory)
```

- Queue of RDD's (voor testen)
- Andere platformen: Kafka, Kinesis, …

Transformations on DStreams

- Meeste wat mogelijk is op RDD kan ook uitgevoerd worden op DStreams
  - UpdateStateByKey
    - Maintain state will continuously receive data
  - Transform()
    - Voor RDD naar RDD functies die niet beschikbaar zijn in Dstreams
  - Window operations
    - Window length
    - Sliding interval



Output / Loads

- To console (print)
- To Files
  - TextFiles
  - ObjectFiles
  - Hadoop Files
- Generiek extern systeem
  - For Each RDD -> wees aandachtig voor waar je de connectie aanmaakt

Load stap van ETL in Spark Streaming

```
In [ ]: def sendRecord(rdd):
            connection = createNewConnection()  # executed at the driver
            rdd.foreach(lambda record: connection.send(record))
            connection.close()

        dstream.foreachRDD(sendRecord)
```

- Gaat dit werken?

Load stap van ETL in Spark Streaming

```
In [ ]: def sendRecord(rdd):
            connection = createNewConnection()  # executed at the driver
            rdd.foreach(lambda record: connection.send(record))
            connection.close()

        dstream.foreachRDD(sendRecord)
```

Verkeerd

- Hier wordt de connectie op de driver aangemaakt en verstuurd naar de nodes
  - Gaat zelden werken

Load stap van ETL in Spark Streaming

```
In [ ]: def sendRecord(record):
            connection = createNewConnection()
            connection.send(record)
            connection.close()

        dstream.foreachRDD(lambda rdd: rdd.foreach(sendRecord))
```

- Is dit beter?

Load stap van ETL in Spark Streaming

```
In [ ]: def sendRecord(record):
            connection = createNewConnection()
            connection.send(record)
            connection.close()

        dstream.foreachRDD(lambda rdd: rdd.foreach(sendRecord))
```

Verkeerd

- Hier wordt er een nieuwe connectie aangemaakt voor elke record
  - Werkt maar zorgt voor veel overhead

Load stap van ETL in Spark Streaming

```
In [ ]: def sendPartition(iter):
            connection = createNewConnection()
            for record in iter:
                connection.send(record)
            connection.close()

        dstream.foreachRDD(lambda rdd: rdd.foreachPartition(sendPartition))
```

- En dit?

Load stap van ETL in Spark Streaming

```
In [ ]: def sendPartition(iter):
            connection = createNewConnection()
            for record in iter:
                connection.send(record)
            connection.close()

        dstream.foreachRDD(lambda rdd: rdd.foreachPartition(sendPartition))
```

Goed

- Hier wordt er een nieuwe connection aangemaakt voor elke partitie
  - Werkt maar zorgt nog steeds voor overhead

Load stap van ETL in Spark Streaming

```
In [ ]: def sendPartition(iter):
            # ConnectionPool is a static, lazily initialized pool of connections
            connection = ConnectionPool.getConnection()
            for record in iter:
                connection.send(record)
            # return to the pool for future reuse
            ConnectionPool.returnConnection(connection)

        dstream.foreachRDD(lambda rdd: rdd.foreachPartition(sendPartition))
```

- En dit?

Load stap van ETL in Spark Streaming

```
In [ ]: def sendPartition(iter):
            # ConnectionPool is a static, lazily initialized pool of connections
            connection = ConnectionPool.getConnection()
            for record in iter:
                connection.send(record)
            # return to the pool for future reuse
            ConnectionPool.returnConnection(connection)

        dstream.foreachRDD(lambda rdd: rdd.foreachPartition(sendPartition))
```

Ideaal

- Hier worden connecties herbruikt waardoor deze niet steeds geopend en gesloten moeten worden

Checkpointing

- Streaming applications blijven draaien (24/7) en moeten dus ook fault-tolerant zijn.
- Hiervoor moeten checkpoints bijgehouden worden in een fout-tolerant opslagsysteem
- Types
  - Metadata checkpointing
  - Data checkpointing

Metadata checkpointing

- Informatie over de streaming berekeningen
  - Configuratie om de applicatie aan te maken
  - De operaties om de applicatie uit te voeren
  - Batches die wachten om verwerkt te worden
- Recover indien de node met de driver van de applicatie uitvalt

Data checkpointing

- Bewaar de gegenereerde RDDs

- Noodzakelijk om een state bij te houden
- Recover indien een verwerkingsnode faalt

```
# enable checkpoints waar "checkpoint" de directory waar ze bijgehouden worden voorstelt
ssc.checkpoint("checkpoint")
```

Accumulators / broadcasting

- Kunnen niet recovered worden via checkpointing
- Maak lazy singletons aan:

```
def getWordExcludeList(sparkContext):
    if ("wordExcludeList" not in globals()):
        globals()["wordExcludeList"] = sparkContext.broadcast(["a", "b", "c"])
    return globals()["wordExcludeList"]

def getDroppedWordsCounter(sparkContext):
    if ("droppedWordsCounter" not in globals()):
        globals()["droppedWordsCounter"] = sparkContext.accumulator(0)
    return globals()["droppedWordsCounter"]
```

Nadelen gebruik DStreams

- Low-level API door gebruik RDD’s ipv DataFrames
- Gebruikt niet de optimalisaties van Dataframes
- Moet casten naar een Dataframe voor de krachtigere high-level API’s te gebruiken zoals Spark SQL

Structured Streaming Spark

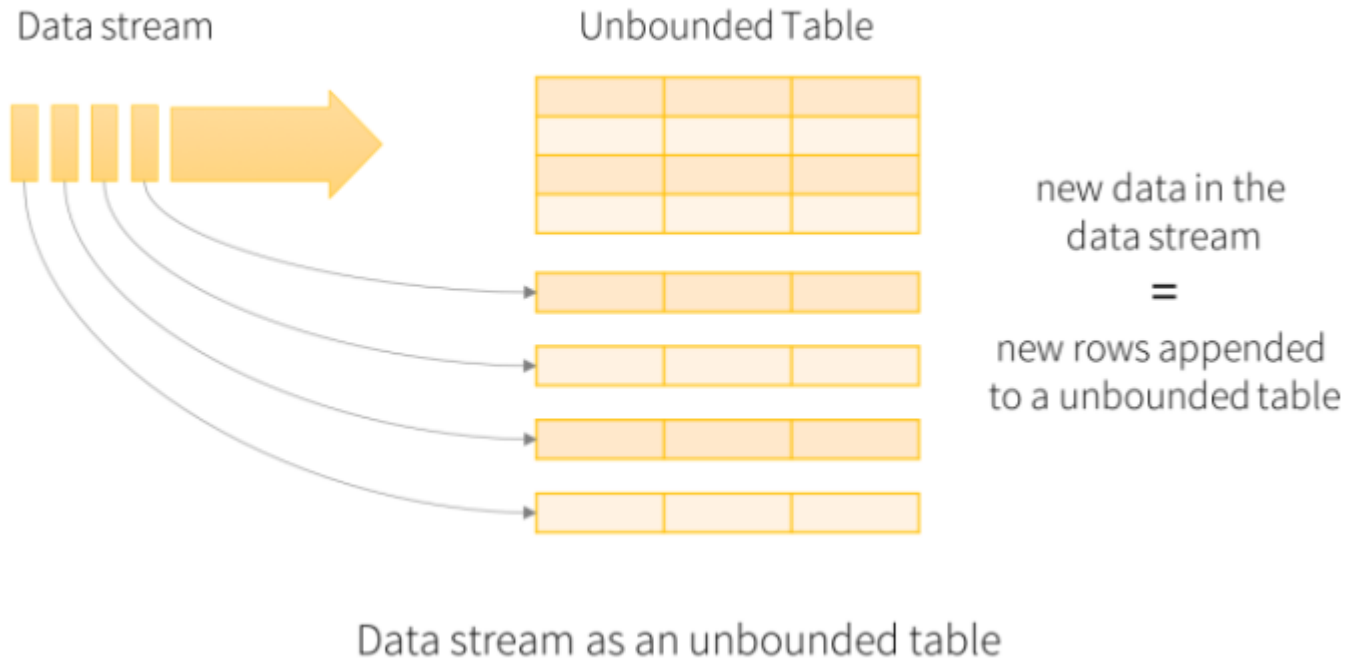
Streaming API direct in DataFrames

- Lost de nadelen van DStreams op

- Maar trager (100ms gegarandeerd vs 10ms niet-gegarandeerd)
- Voor sommige applicaties minder flexibel

Verschillen: Geen concept van batch

- Nieuwe data wordt toegevoegd aan een dataframe

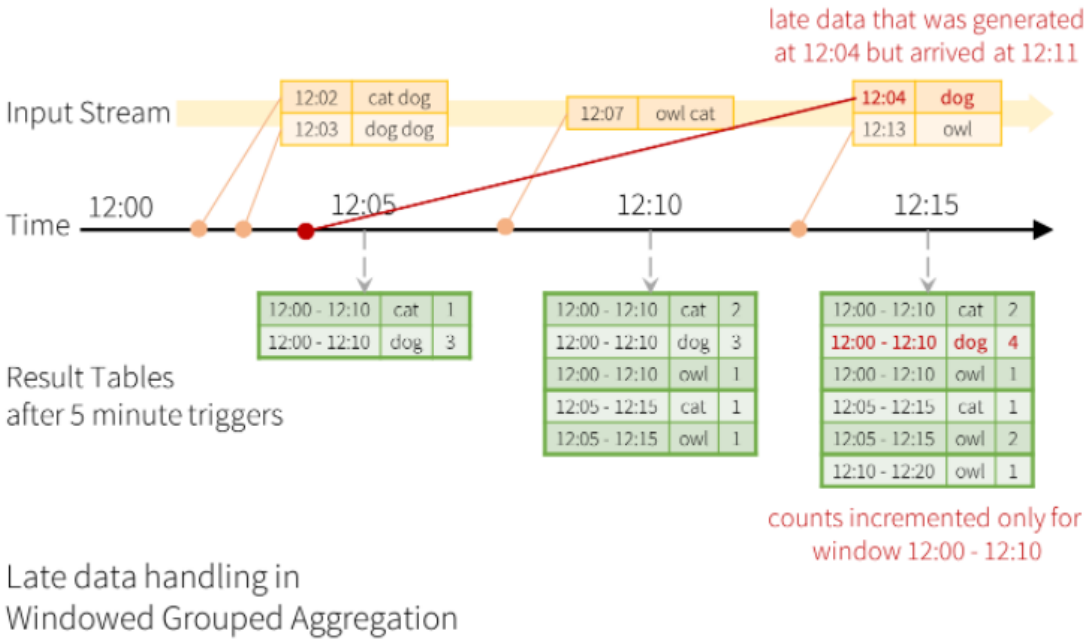


Input sources

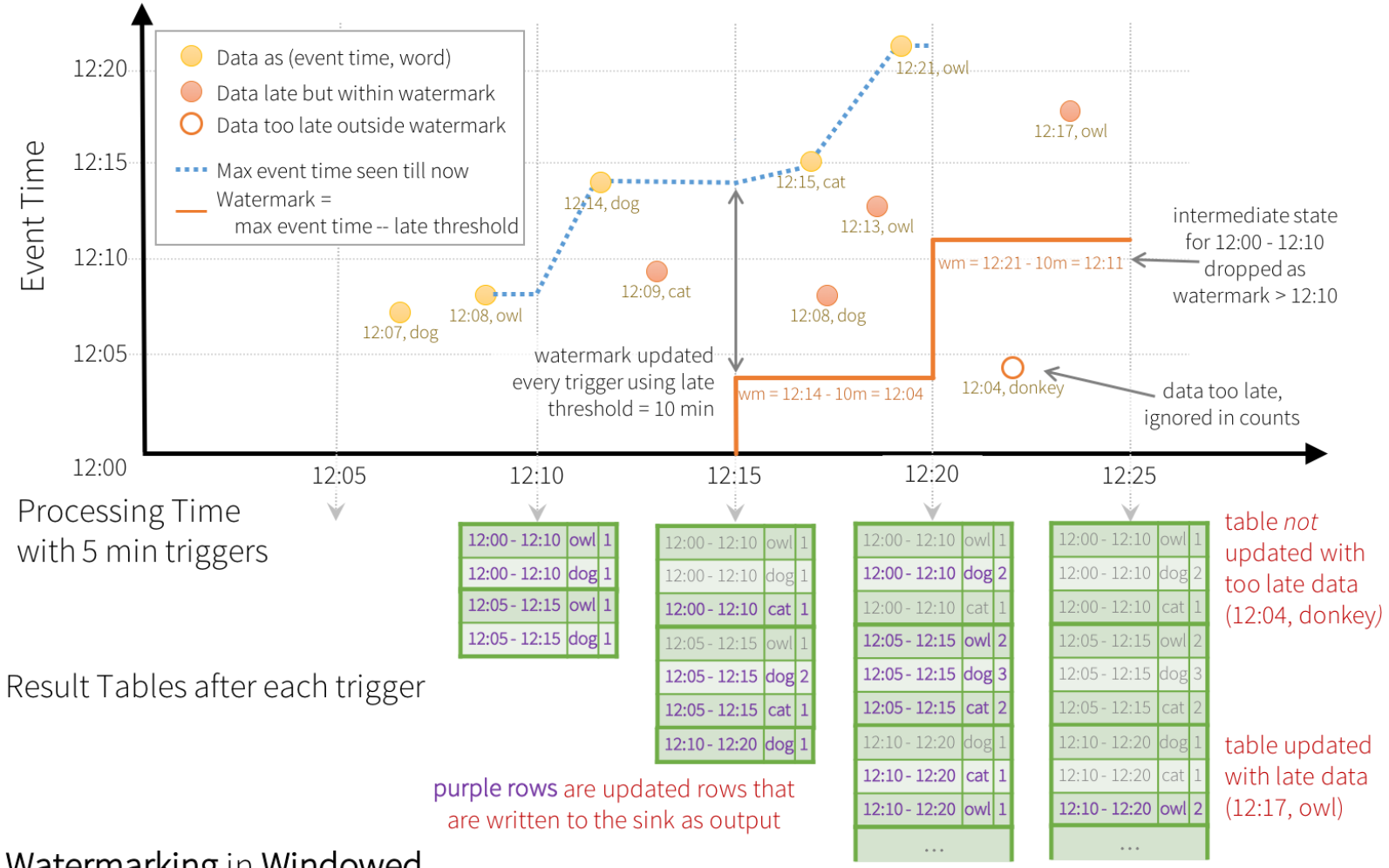
- Files
- Kafka
- Socket (voor testing)
- Rate source
  - Genereer data aan een bepaalde snelheid

Transformaties

- Alles wat mogelijk is op Dataframes kan hier gebruikt wordt
  - Groupby
  - Window
    - Heeft timestamp kolom nodig
  - Late data



Watermark om te late data te negeren



Watermarking in **Windowed Grouped Aggregation** with **Update Mode**

Structured streaming with MLlib

```
from pyspark.ml import PipelineModel
from pyspark.sql.functions import udf
from pyspark.sql.types import DoubleType

# Load the trained MLlib model
model = PipelineModel.load('model_path')

# Define a function to apply the model to a batch of data
def predict_batch(batch_df):
    # Apply the model to the batch
    predictions = model.transform(batch_df)
    # Extract the prediction values from the output column
    get_prediction = udf(lambda x: x[1], DoubleType())
    prediction_values = predictions.select(get_prediction('probability')).collect()
    # Print the predictions for this batch
    print(prediction_values)

# Create a structured stream from a Kafka topic
stream = spark.readStream.format('kafka').option('kafka.bootstrap.servers', 'localhost:9092').option('subscribe', 'input_topic').load()

# Apply any necessary transformations to the stream
processed_stream = preprocess(stream)

# Use foreachBatch to apply the MLlib model to each batch of data in the stream
predictions = processed_stream.writeStream.foreachBatch(predict_batch).start()

# Start the streaming query
predictions.awaitTermination()
```

Load

- Output modes
  - Append -> new rows written
  - Update -> updated rows
  - Complete -> complete df

Load

- Sinks
  - File
  - Kafka
  - Foreach
  - Console (debugging)
  - Memory (debugging)

- Niet alle sinks zijn even fout tolerant en niet alle queries laten alle output modes toe
- https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html