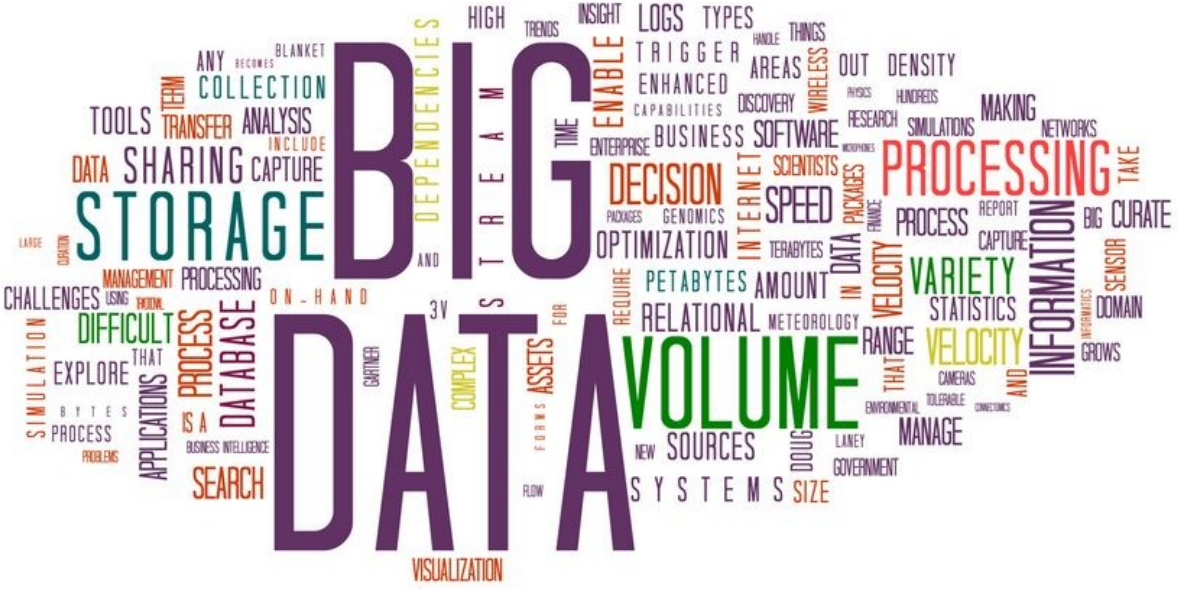
Session 3

- Structured streaming Spark
- More about Spark
  - RDDs
  - more about functions
  - unions and joins
  - aggregations (like GroupBy)
- Introduction to Kafka
- Introduction to research methods
  - design science research
  - the programming projects
- Exercises 2 & 3



Structured Streaming Spark

Limitations of Hadoop MapReduce

- *Batch*-oriented, with jobs that can take day(s)
- Heavily *disk* based
  - less suitable for *iterative* and *interactive* tasks
- *Rigid*, with few operations, e.g., for
  - structured data schemas and optimisation (e.g., SQL)
  - sharing data (broadcasts and accumulators)
  - machine learning
  - graph processing
  - *streaming (or live) data*

Structured Streaming Spark

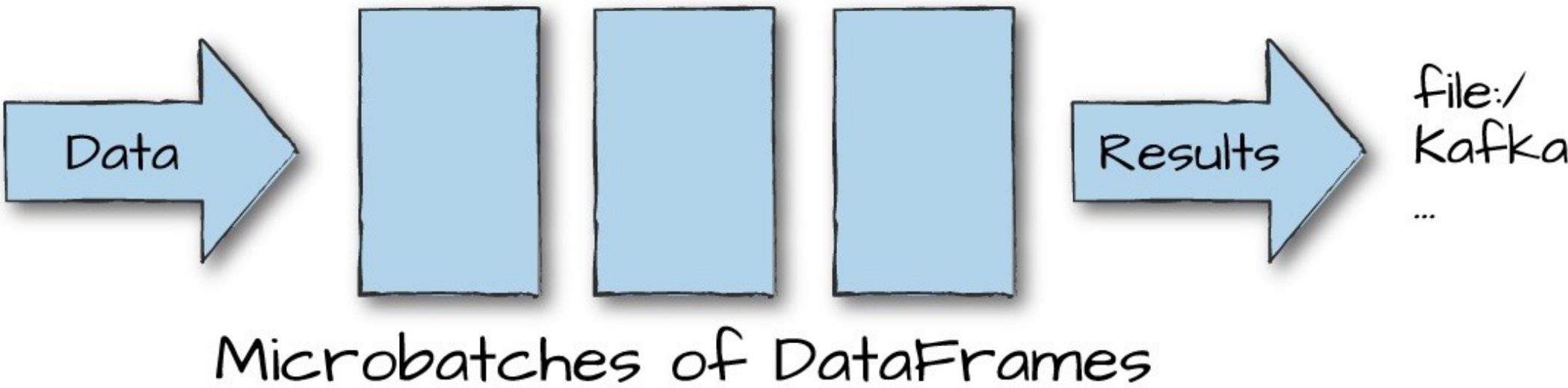
- «Plain» Streaming Spark:
  - Spark 1.x, based on RDDs
- Structured Streaming Spark:
  - Spark >2.0, based on DataFrames (and DataSets, SQL, ...)
  - built on the Structured Spark engine
  - streaming computations on dynamic data
    - expressed similarly to batch computations on static data
    - running jobs incrementally
    - updating results continuously as streaming data arrive
  - Based on the standard Dataset/DataFrame/SQL operations
    - APIs in Scala, Java, Python, R...

Structured Streaming Spark

- Stream processing engine:
  - fast, scalable, reliable and fault-tolerant
  - end-to-end and exactly-once guarantees
  - checkpointing, replayable sources, write-ahead logs
  - without the user having to reason about streaming.
- Supports:
  - streaming aggregations
  - event-time windows
  - stream-to-batch and stream-to-stream joins

Processing

- Micro-batch processing:
  - the default
  - processes data streams as a series of small batch jobs
  - end-to-end latencies as low as 100 milliseconds
  - exactly-once fault-tolerance guarantee
- Continuous processing:
  - since Spark 2.3
  - low-latency processing mode
  - end-to-end latencies as low as 1 millisecond
  - at-least-once guarantees
- Uses the standard Dataset/DataFrame/SQL operations

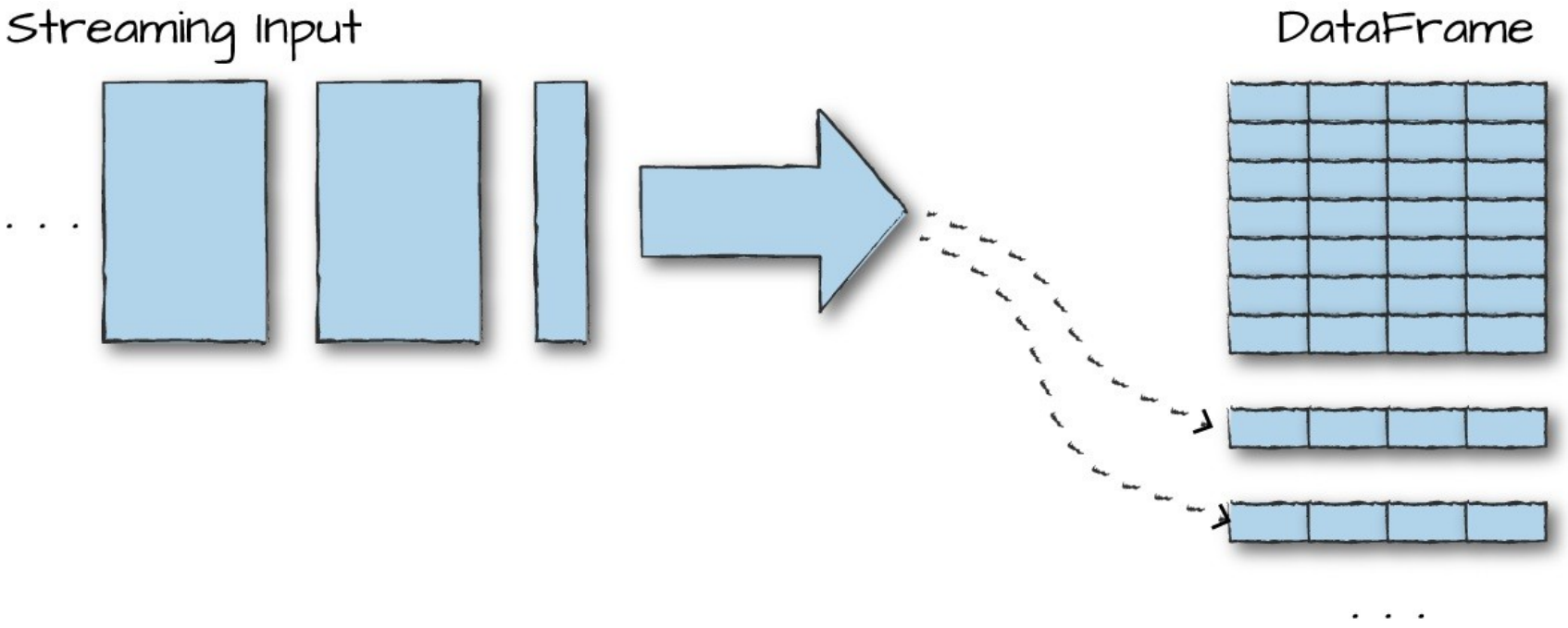

Data
Results
file:/
Kafka
...
Microbatches of DataFrames

State store

- State store:
  - versioned key-value store that provides both read and write operations
  - handles stateful operations across batches
- Two built-in state store providers:
  - HDFS state store provider
    - all the data is stored in memory map in the first stage
    - then backed by files in an HDFS-compatible file system
  - RocksDB state store implementation

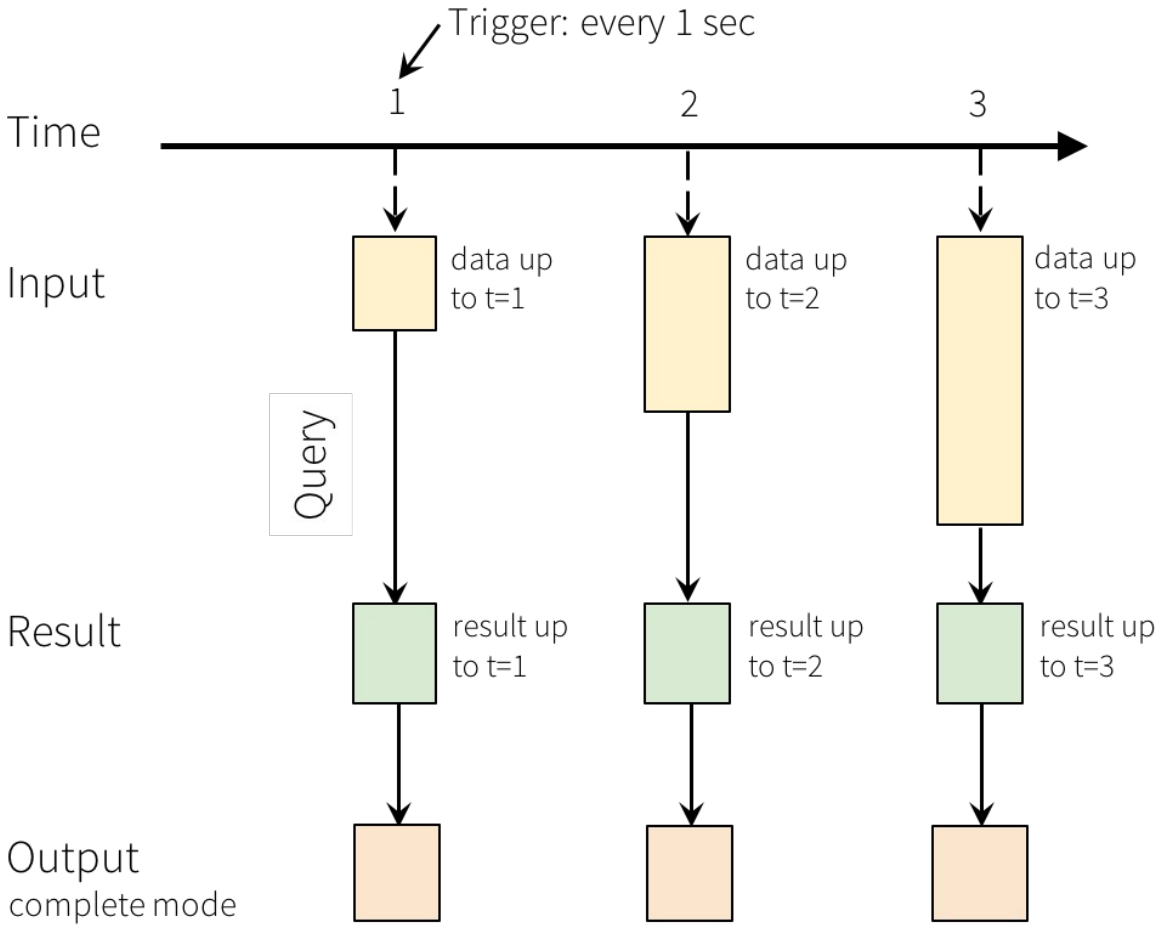
Streaming input

- Consider the input data stream as the input DataFrame
- Every data item that is arriving on the stream is a new row appended to the bottom of the Input DataFrame



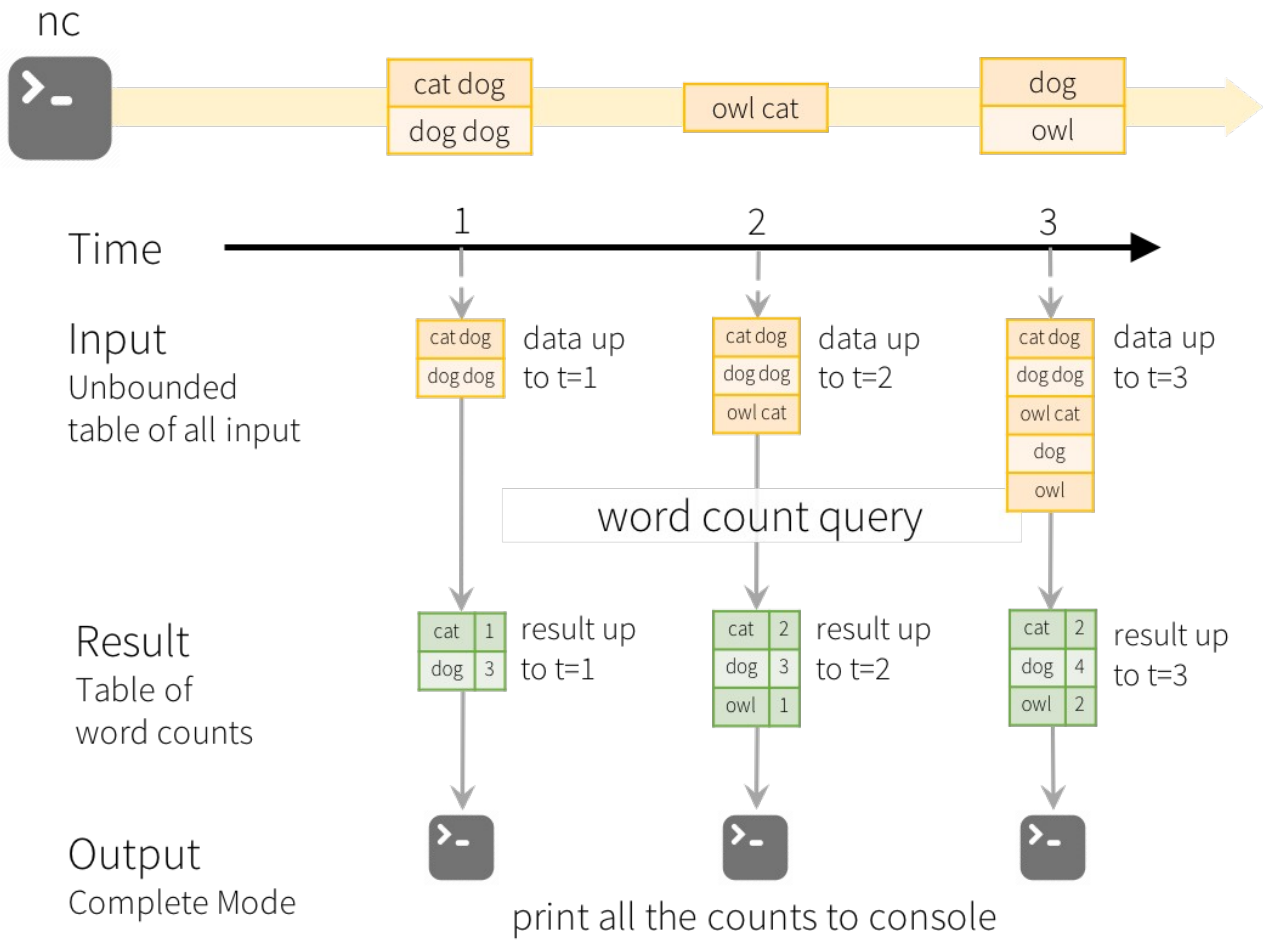
Streaming output

- Streaming Spark:
  - treats the input data stream as a table that grows continuously
  - runs an incremental job on the unbounded input table
    - expressed similarly to a job on a static input table
  - treats the output data stream as a table that is repeatedly being
    - appended to (append mode, the default)
    - overwritten (complete mode)
    - updated (update mode)
- Triggers determine how often the incremental job is run
  - the default is as often as possible



Programming Model for Structured Streaming

Example



Model of the Quick Example

Programming model

- Transformation of the Input DataFrame generates the Output DataFrame
  - every trigger interval (e.g., every 1 second):
    - new rows get appended to the Input DataFrame,
    - which eventually updates the Output DataFrame
  - whenever the output frame changes, the results are written to a sink
  - output modes:
    - complete: the entire updated Output DataFrame is written to the sink
    - append: only newly appended rows since the last trigger are written
      - applicable when existing rows are not expected to change
    - update: only outputs rows changed since the last trigger are written
      - equivalent to append if no aggregations
    - *each mode is applicable on certain types of queries...*

Processing details

- *Structured Streaming does not materialize the entire table:*
  - reads the latest available data from the streaming data source
  - processes it incrementally to update the result
  - then discards the source data
- Only keeps around the minimal intermediate state data
  - such as intermediate counts required to update an output grame
- High-level model, transparently takes care of:
  - running aggregations
  - reliability, fault-tolerance
  - ensuring consistency (at-least-once, or at-most-once, or exactly-once)
  - updating the Output DataFrame when there is new data

Handling event times

- Event time:
  - the time embedded in the data itself
  - often preferred to the time when Spark receives them
  - natural representation:
    - each event from the devices is a row in the table
    - each event-time represented as a column value in the row
  - allows window-based aggregations (e.g. number of events every minute) as a special type of grouping and aggregation on the event-time column
    - each time window is a group
    - each row can belong to multiple windows/groups
  - event-time-window-based aggregation queries can be defined consistently on both a static dataset as well as on a data stream

Handling late data

- Late data:
  - delayed arrival into to Spark
  - «data that arrive later than expected based on its event-time»
  - data about older events are appended below more recent ones
- Can be handled using update mode:
  - existing aggregates in the Output DataFrame are updated with late data
  - the user can specify the threshold for late data
    - the engine cleans up old intermediate state automatically

Fault Tolerance Semantics

- End-to-end exactly-once fault tolerance:
  - the same input row effects the Output DataFrame exactly once
  - one of key goals behind the design of Spark Structured Streaming
  - reliably tracking the exact progress of the processing
  - can handle any kind of failure by restarting and/or reprocessing
    - replayable sources
- Streaming sources assumed to have offsets (similar to Kafka offsets) to track the read position in the stream.
  - checkpointing and write-ahead logs record the offset range of the data being processed in each trigger
  - streaming sinks are designed to be idempotent for handling reprocessing

Structured Streaming in Pyspark

Data sources in Pyspark

- Streaming DataFrames through the DataStreamReader interface:
  - SparkSession.readStream()
    - i.e., spark.readStream()
  - specify the details in a way similar to static DataFrames
    - i.e., data format, schema, options...
- Input from streams:
  - streaming_df = spark.readStream.format(format).load(location)
  - core streaming sources:
    - socket, folder, HDFS, Kafka

Built-in streaming sources

- File source:
  - reads files written in a directory as a stream of data
  - files processed in the order of file modification time
  - supported file formats: text, CSV, JSON, ORC, Parquet...
- Kafka source:
  - reads data from Kafka
- Socket source (for testing):
  - reads utf-8 text data from a socket connection
  - the listening server socket is at the Spark driver
  - *does not guarantee end-to-end fault-tolerance*
- Rate sources (for testing):
  - generate data at the specified number of rows per second

Data sinks in Pyspark

- Streaming DataFrames through the DataStreamWriter interface:
  - DataFrame.writeStream()
    - i.e., streaming_df.writeStream()
- Output to streams:
  - streaming_df.writeStream. … .start() # usual streaming action
    streaming_df.awaitTermination()
  - core streaming sinks:
    - socket, console, memory, foreach(Batch), folder, HDFS, Kafka

Data sinks in Pyspark

- Streaming DataFrames through the DataStreamWriter interface:
  - specify the details in a way similar to static DataFrames
    - i.e., data format, options...
  - also streaming specific:
    - outputMode: what gets written to the output sink.
    - queryName (optional): specify a unique query name
    - trigger interval (optional): default is as fast as possible
    - checkpoint location: where to write checkpoint information
  - other methods:
    - foreach: custom write logic on every output row
    - foreachBatch: custom write logic on the output of each micro-batch

Built-in streaming sinks

- File sink; stores the output to a directory.
- Kafka sink: stores the output to one or more topics in Kafka.
- Foreach(Batch) sink: runs arbitrary computation on the records in the output
- Console sink (for debugging):
  - prints the output to the console/stdout every time there is a trigger
  - both append and complete modes are supported
  - only low data volumes as the entire output is collected and stored in the driver's memory after every trigger.
- Memory sink (for debugging):
  - output is stored in memory as an in-memory table
  - modes and data volumes as for console sink

StreamingQuery

- Lazy evaluation of streaming transformations
- Usual streaming action: DataStreamWriter.start()
  - returns a StreamingQuery object
  - executes continuously in the background as new data arrives
  - call awaitTermination() on the StreamingQuery object
- Important methods:
  - awaitTermination([timeout])
    - waits for the termination of this query by query.stop() or an exception
  - exception() - halts the streaming query and throws exception
  - explain([extended]) - prints the execution plans to the console
  - stop() - stops the streaming query