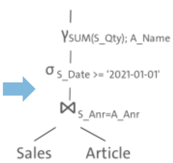
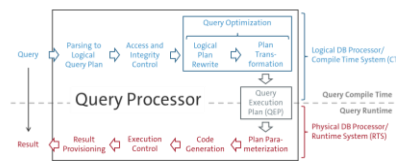
Summary

- Architecture of Database Systems
- Transaction Management
- Modern Database Technology
- Data Warehouses and OLAP
- Data Mining
- Big Data Analytics

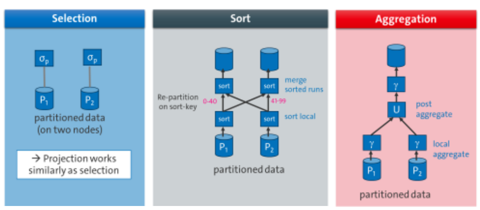
Recap Query Processing

```
SELECT A_Name, SUM(S_Qty)
FROM Article, Sales
WHERE A_Anr = S_Anr AND
      S_Date >= '2021-01-01'
GROUP BY A_Name
```





Intra-Operator Parallelism



Parallel Joins: Symmetric Repartitioning Asymmetric Repartitioning
Fragment and Replicate

Transparency for Query Formulation

Fragmentation transparency

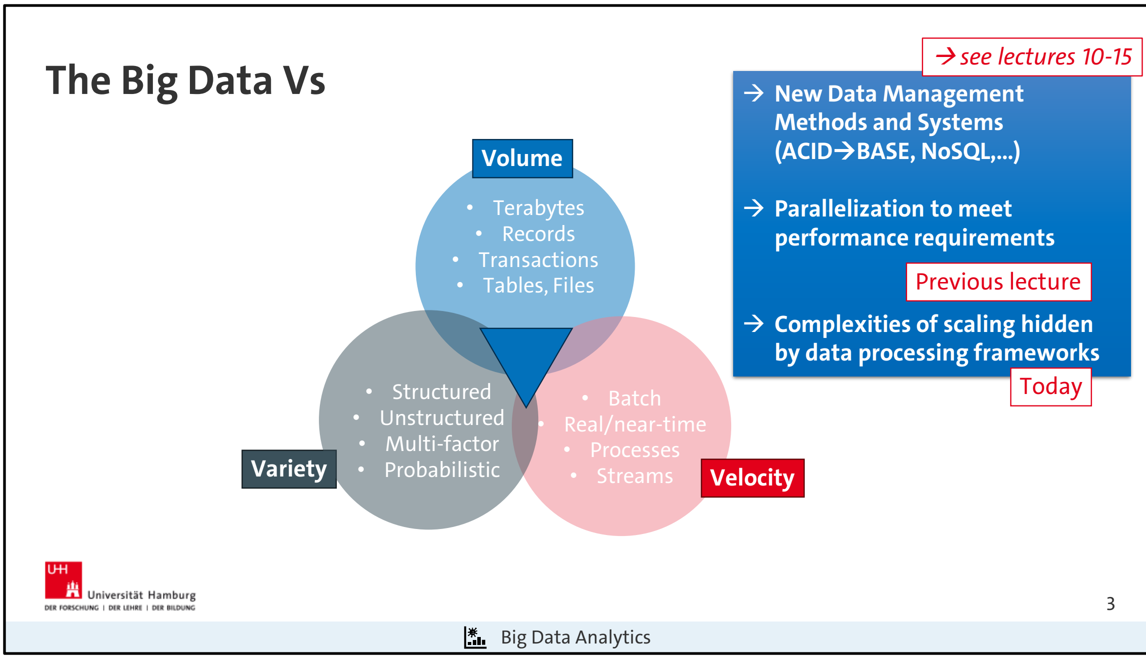
```
select COUNT(*)
  from Orders
 where Amount>=5
```

Network transparency

```
select COUNT(*)
  from (Orders₁@Node₁
        union
        Orders₂@Node₂)
 where Amount>=5
```

Allocation transparency

```
select COUNT(*)
  from (Orders₁ union Orders₂)
 where Amount>=5
```



More Vs:

- *Veracity*: Isyourdata(source) trustworthy/ meaningful?
- *Visualization*: Howtocommunicate? insights& knowledge?
- *Value*: Howtousedatafor(machine) learning, optimization, ...?
- (Volatility, Vulnerability, Validity, ...)

Complexities of scaling beyond plain operator parallelization

- Monitoring (health checks, application statistics
- Scheduling (e.g. rebalancing)
- Fault-tolerance, e.g. restarting workers, rescheduling failed work

Batch & Stream Processing

Batch

- Operates on complete data
- Periodic jobs, e.g. during night times

→ Efficient but high latency

Volume

Stream

- Operates on partial data (one-at-a-time)

→ Low end-to-end latency

Challenges

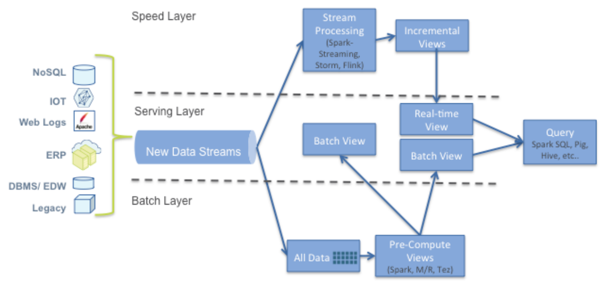
- Long-running jobs
- Data may arrive delayed or out-of-order
- Incomplete input

Velocity

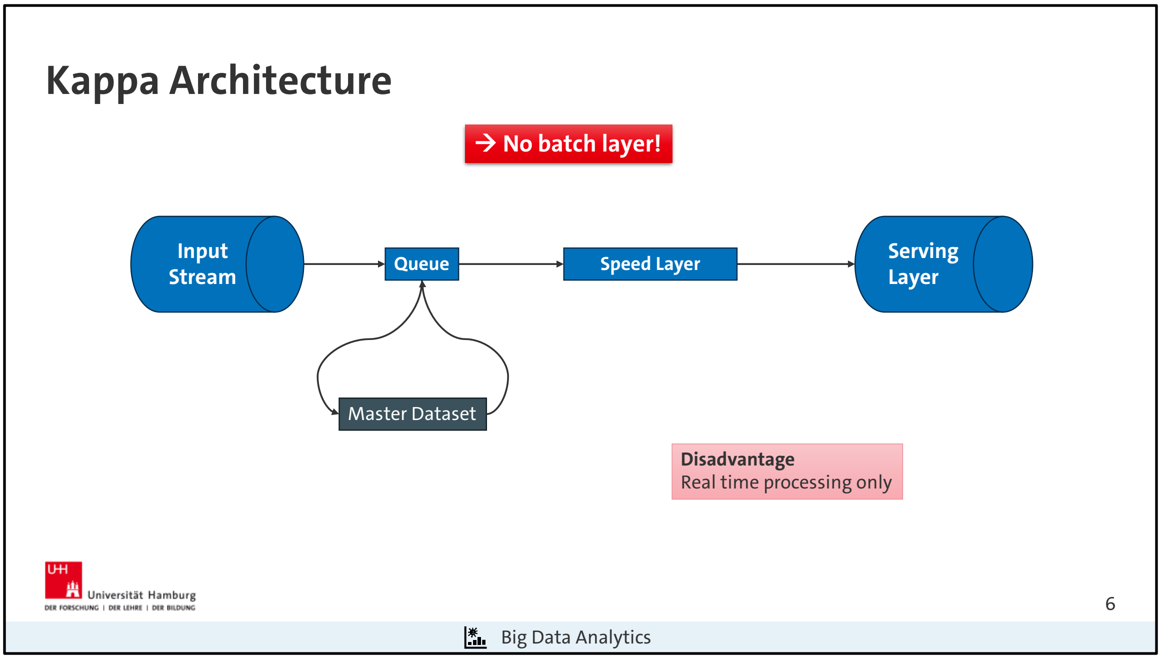
Lambda Architecture

→ *see lecture 16*

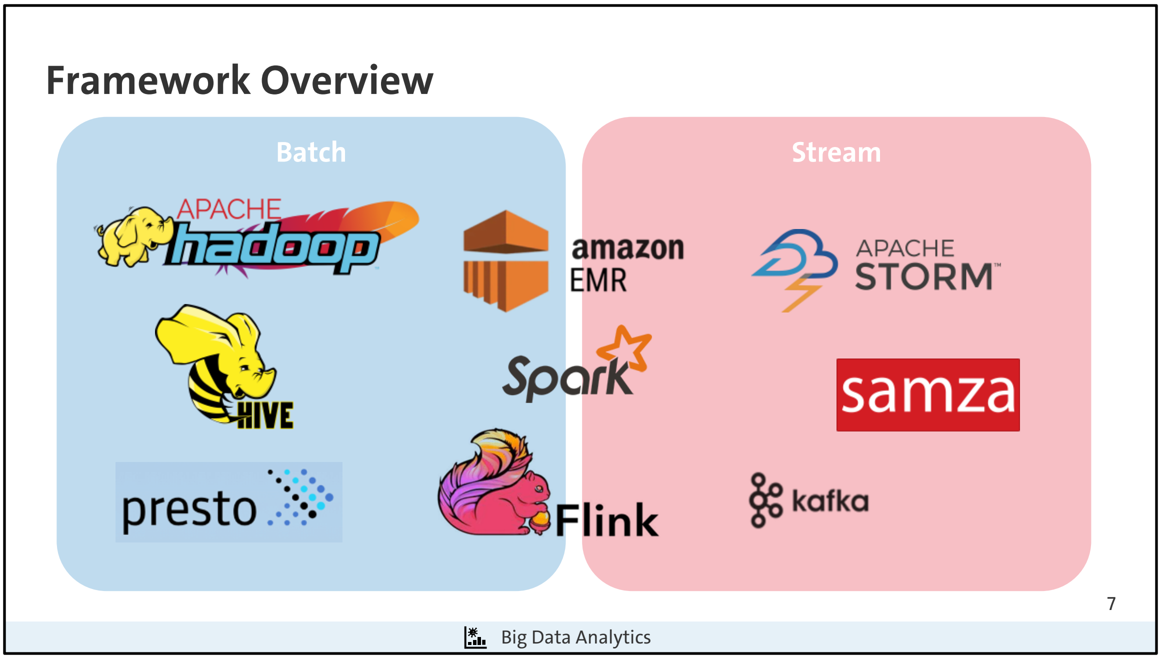
→ Batch & Stream processing



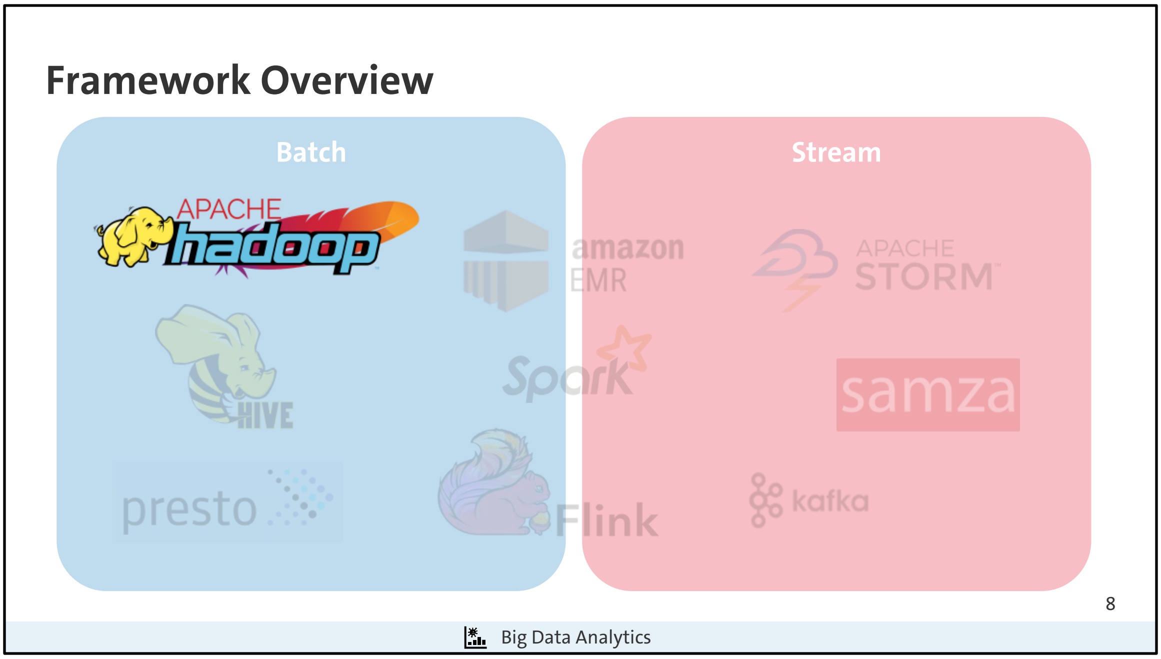
Disadvantage
2 code bases & 2 deployments, e.g. Hadoop & Storm



Introduces backpressure (without large enough buffer)

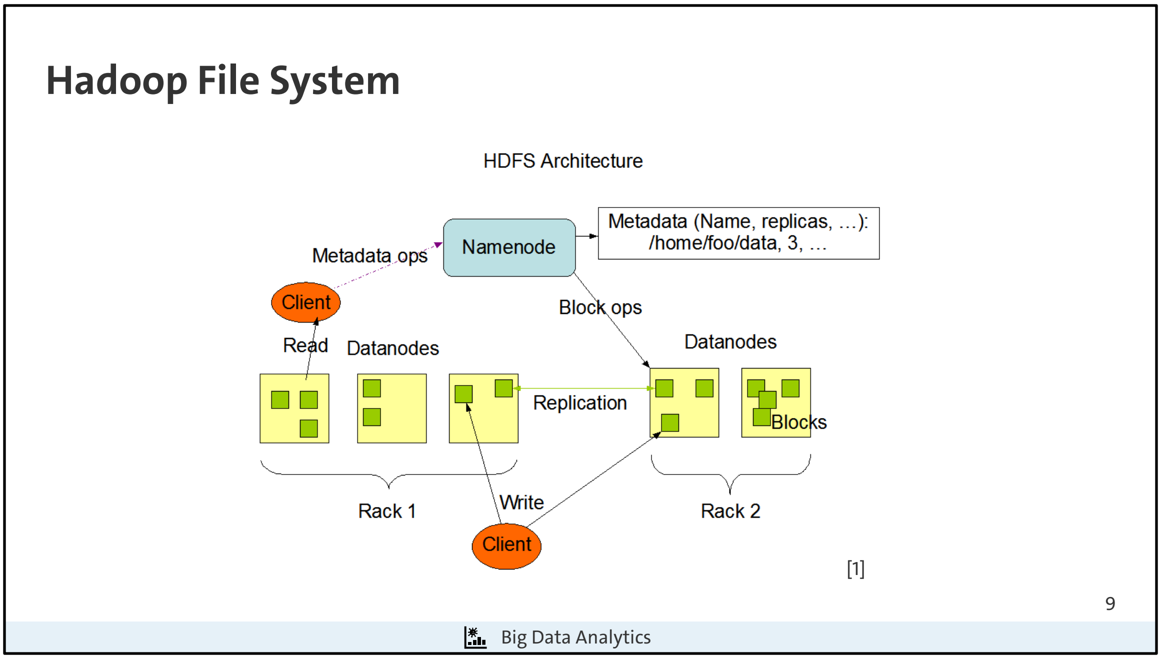


- EMR – Elastic Map reduce, provided as web service, provides managed Hadoop, spark, presto
- Presto – Distributed (SQL) query engine, allows using different data sources, e.g. Hadoop, kafka, MongoDB, MySQL,…
- Kafka → Kappa architecture



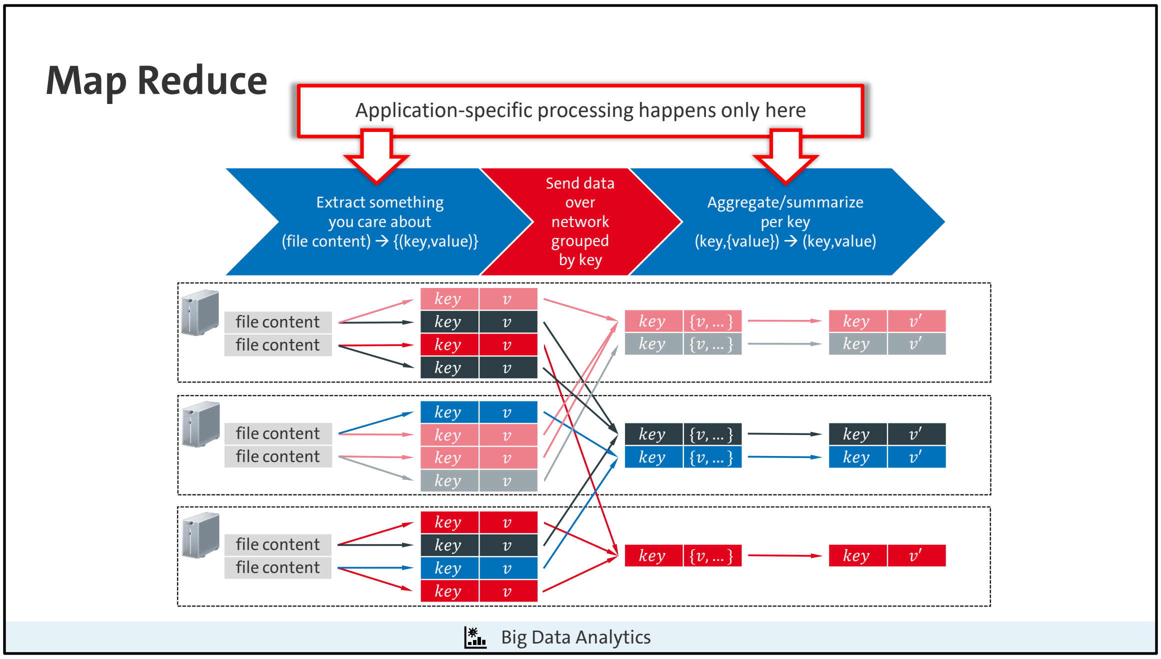
- Uses the Map-Reduce Paradigm
- HDFS: Scalable , shared nothing file system for throughput-oriented workloads

- Other Hadoop projects
    - Hive : SQL(-dialect) compiled to YARN jobs (Facebook)
    - Pig : workflow-oriented scripting language (Yahoo)
    - Mahout : Machine Learning algorithm library in Map Reduce
    - Flume : Log Collection and processing framework
    - Whirr : Hadoop provisioning for cloud environments
    - Giraph : Graph processing à la Google Pregel
    - Drill , Presto, Impala : SQL Engines



- Modelled after: Googles GFS (2003)
- Cluster Nodes:
  - Single namenode: Metadata (files + block locations)
  - Single master server: Manages file system namespace and regulates access to files by clients
  - Multiple datanodes: Save fileblocks (usually 64 MB), blocks replicated for fault tolerance and read performance → Data can be used where it is stored, usually one data node per physical cluster node, serve read and write requests
- Design goal: Maximum Throughput and data locality for Map-Reduce

[1] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, accessed 02.07.2024

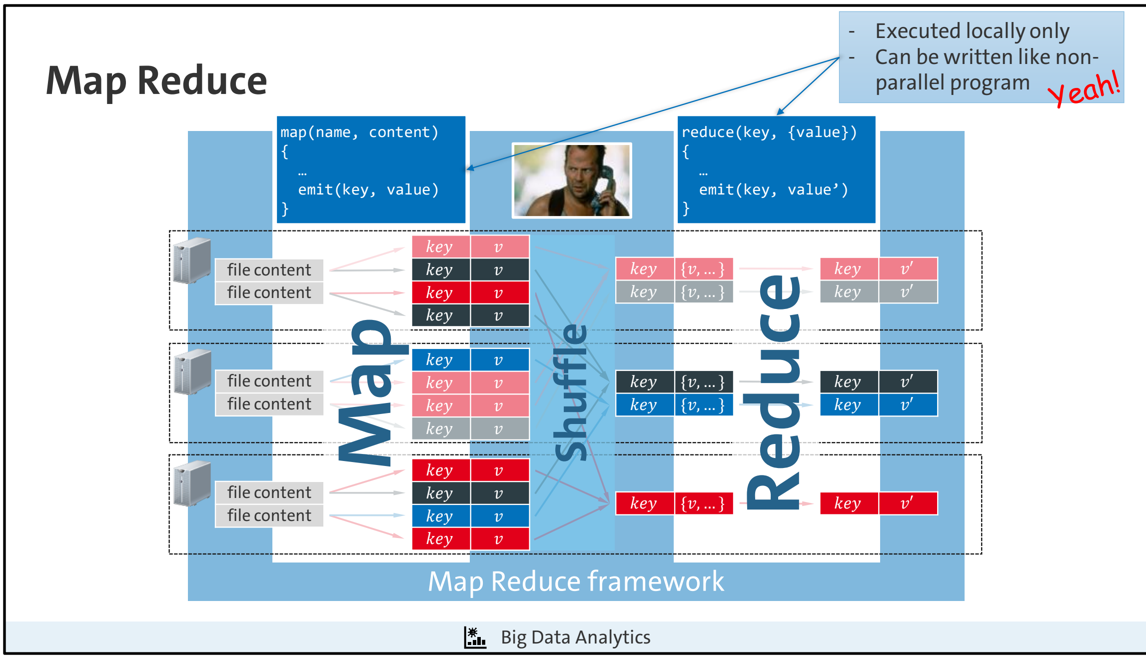


Data Foundation

- Key-Value Pairs (key, value)
- Key and Values can be everything

Map Reduce

- Framework for parallel computing
- Programmers get simple API
- Do not have to worry about
    - Parallelization
    - Data distribution
    - Load balancing
    - Fault tolerance
- Allows everybody to process huge amount of data (terabytes/petabytes/...) on thousands of computer nodes



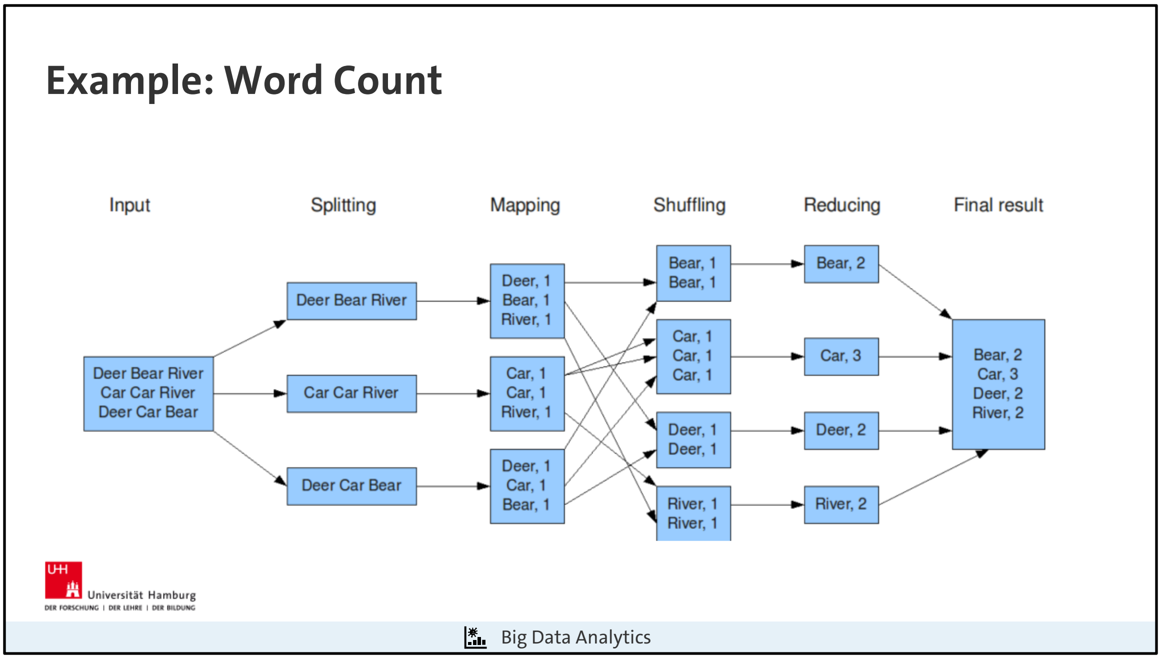
- The programmer essentially only specifies two (sequential) functions
- Framework takes care of all parallelization
- Ships code of work nodes
- Performs shuffle step
- Monitors worker progress
- Handles node failures

Map: map(k1, v1) → list(k2, v2)

- Inputs data record and outputs a set of intermediate key-value pairs, each of type k2 and v2
- Types can be simple or complex user-defined objects
- Each map call is fully independent (no execution ordering, sync or communication)

Reduce: reduce(k2, list(v2)) → list(k3, v3)

- Combines information across records that share the same intermediate key
- Each reduce call is fully independent



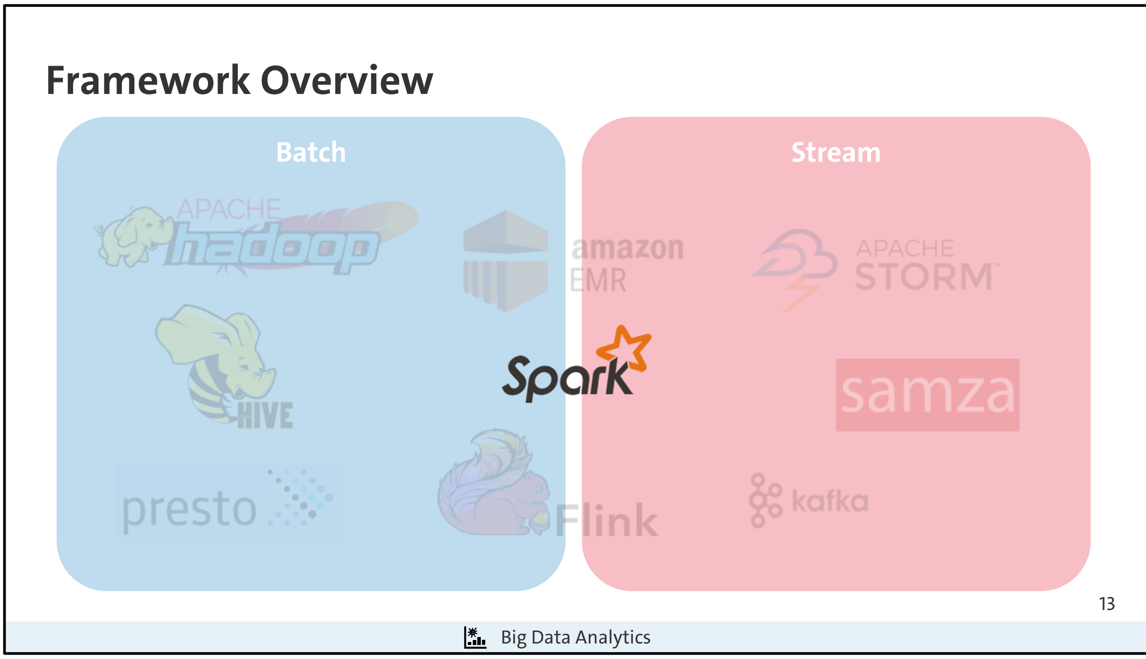
Map Reduce Limitations

API

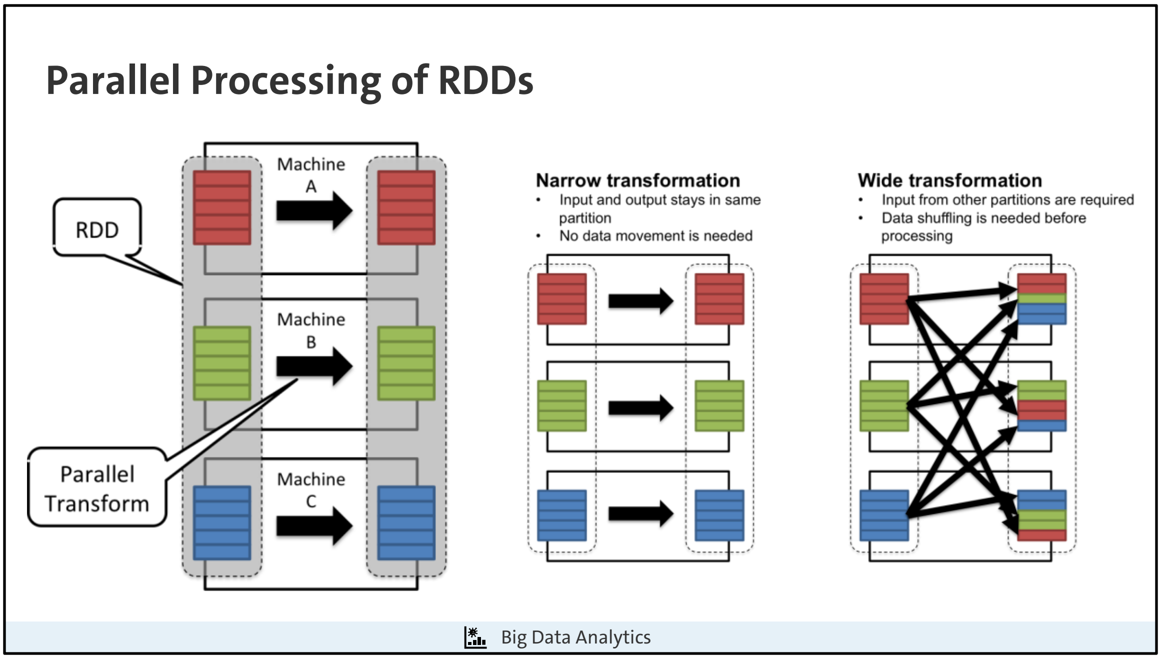
- Very simple
- Not every problem necessarily fit into map-reduce abstraction
- Does not compose very well into larger programs
- No support for cyclic programs
- Fixed data flow

Performance

- Pure disk-based data processing implies performance bottle necks in larger programs
- Loops write state complete to disk in every iteration (might be peta bytes of disk IO each time)

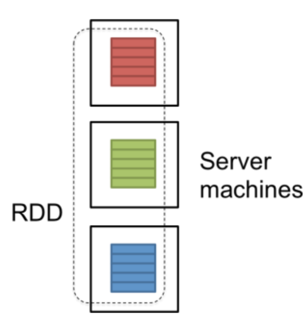


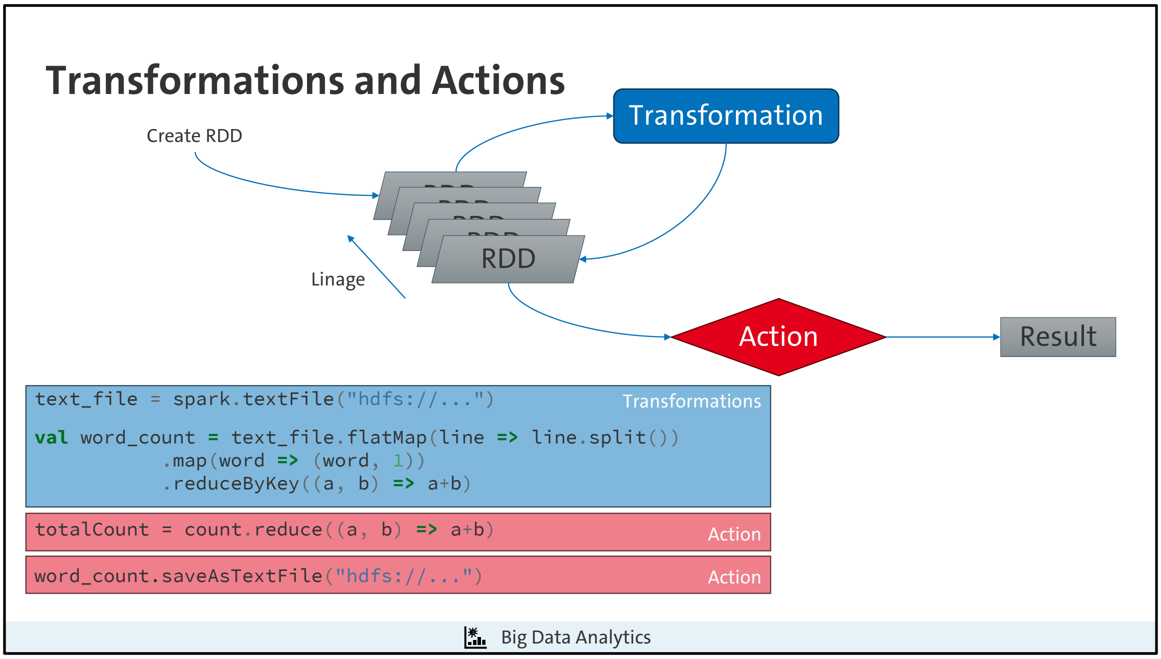
- Basic idea: „In-Memory“ Hadoop optimized for iterative processing (e.g. k-means)
- Resilient Distributed Datasets (RDDs): partitioned, in-memory set of records



Resilient Distributed Datasets (RDDs)

- Sparks original in-memory data structure
- Fault-tolerant immutable distributed collection of data
- Strongly typed but not necessarily structured like a table, e.g. a collection of documents
- No schema imposed
- Manipulated with functional programming constructs in low-level transformation and actions
- Two ways to create RDDs
    - parallelizing an existing collection in your driver program
    - referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, etc.



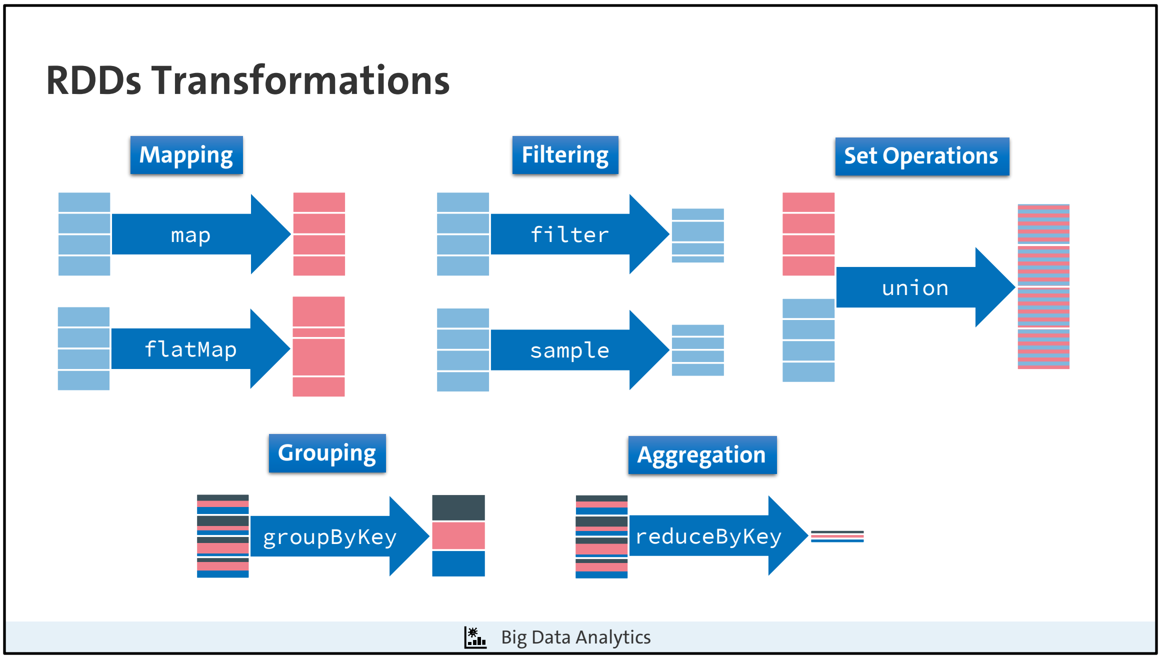


Transformations

- (High-order) functions to create an RDD/Dataset from existing RDDs/Datasets
- Just build up the data flow DAG when executed in the driver program (nothing happens with the data)

Actions

- Return a value (actual data) to the driver program (or write out data)
- Trigger execution of whole data flow DAG
- Examples: collect, count, reduce, countByKey, saveAsTextFile, saveAsObjectFile
- List of actions: http://spark.apache.org/docs/latest/programming-guide.html#actions



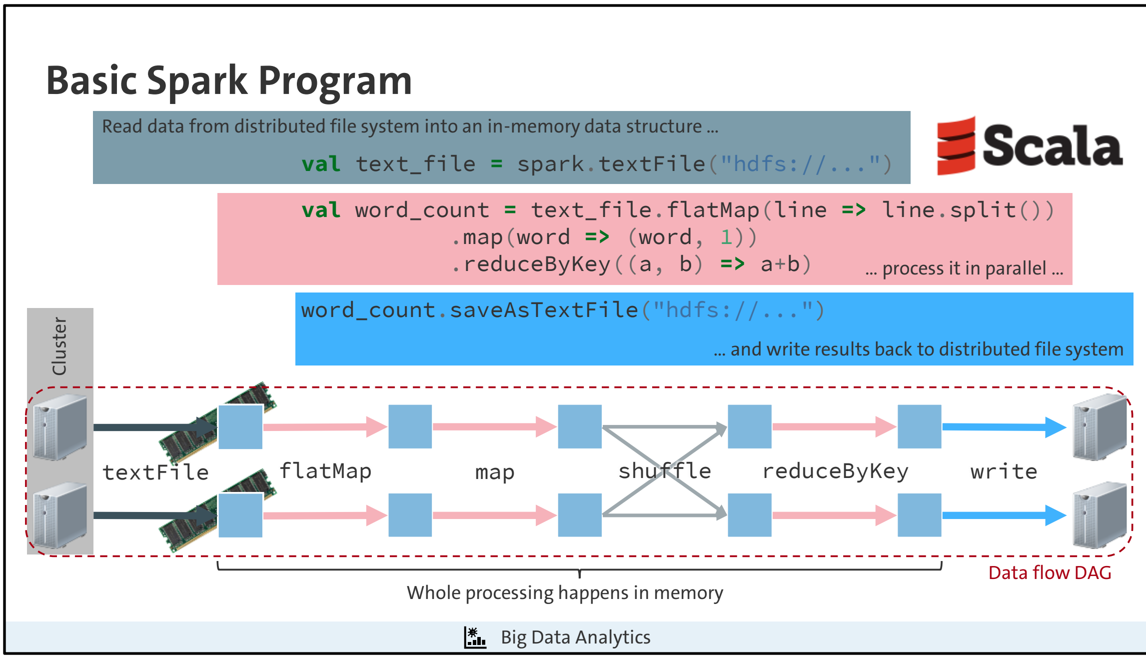
- **map**(*func*): Return a new distributed dataset formed by passing each element of the source through a function *func*.
- **flatMap**(*func*): Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
- **filter**(func): Return a new dataset formed by selecting those elements of the source on which func returns true.
- **sample**(withReplacement, fraction, seed): Sample a fraction fraction of the data, with or without replacement, using a given random number generator seed.
- **union**(otherDataset): Return a new dataset that contains the union of the elements in the source dataset and the argument.
- **groupByKey**([numPartitions]): When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
- **reduceByKey**(func, [numPartitions]): When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.

- List of all transformations: http://spark.apache.org/docs/latest/programming-guide.html#transformations

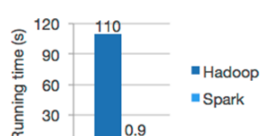
RDD Transformation Exercise

Which transformations and/or actions can be used to implement the following scenarios?

1. Count all items in a basket that belong to the item group “food”
2. Create a histogram
3. Create an estimated histogram for a very large dataset, i.e. where reading the whole dataset for creating the histogram would take too long
4. Calculate the end-of-year bonus (based on yearly salary) from a table containing the monthly salaries.
5. Rename a department



- Generalization of Map Reduce
  - No specialization
  - Support for wide range of applications in a single engine
  - Map Reduce is just one set of supported constructs
- Two major improvements
  - Keeps data memory while processing
  - Generalizes data flow to DAGs and lets user define data flow
- Key features
  - Handles batch, interactive, and real-time processing within a single framework
  - Native integration with Java, Python, and Scala
  - Programming at a higher level of abstraction



Logistic regression in Hadoop and Spark
[http://spark.apache.org/]

Language Integrations

Scala, Java, and Python

```scala
val text_file = spark.textFile("hdfs://...")
val word_count =
  text_file.flatMap(line => line.split())
           .map(word => (word, 1))
           .reduceByKey((a, b) => a+b)
word_count.saveAsTextFile("hdfs://...")
```

Scala

```java
JavaRDD<String> text_file = spark.textFile("hdfs://...")
JavaRDD<Integer> word_count =
  text_file.flatMap(line -> line.split())
           .map(word -> (word, 1))
           .reduceByKey((a, b) -> a+b)
word_count.saveAsTextFile("hdfs://...")
```

Java

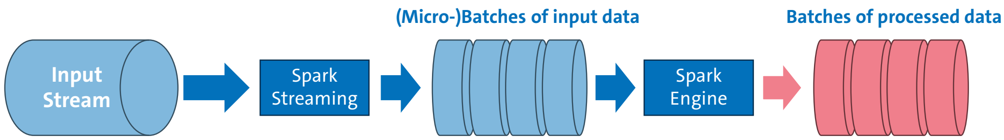
```python
text_file = spark.textFile("hdfs://...")
word_count =
  text_file.flatMap(lambda line: line.split())
           .map(lambda word: (word, 1))
           .reduceByKey(lambda a, b: a+b)
word_count.saveAsTextFile("hdfs://...")
```

python™

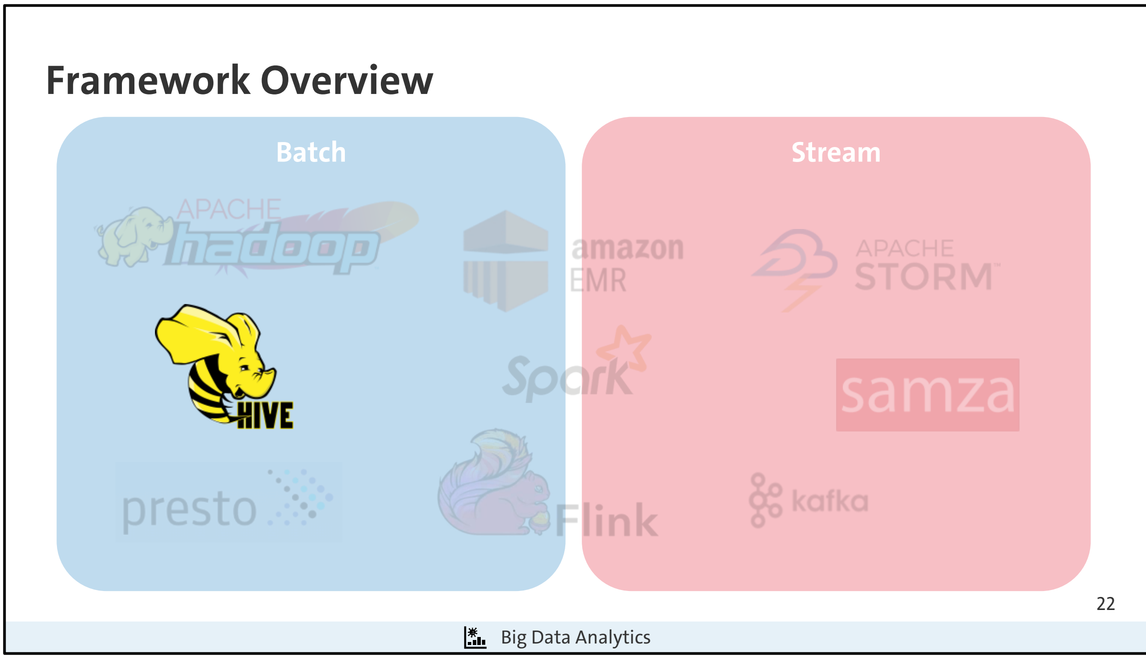
There's also an API for R

Spark Streaming

Split stream into micro-batches to process with Spark engine



DStream: Discretized RDD
→ RDDs are processed in order
but no ordering within RDD



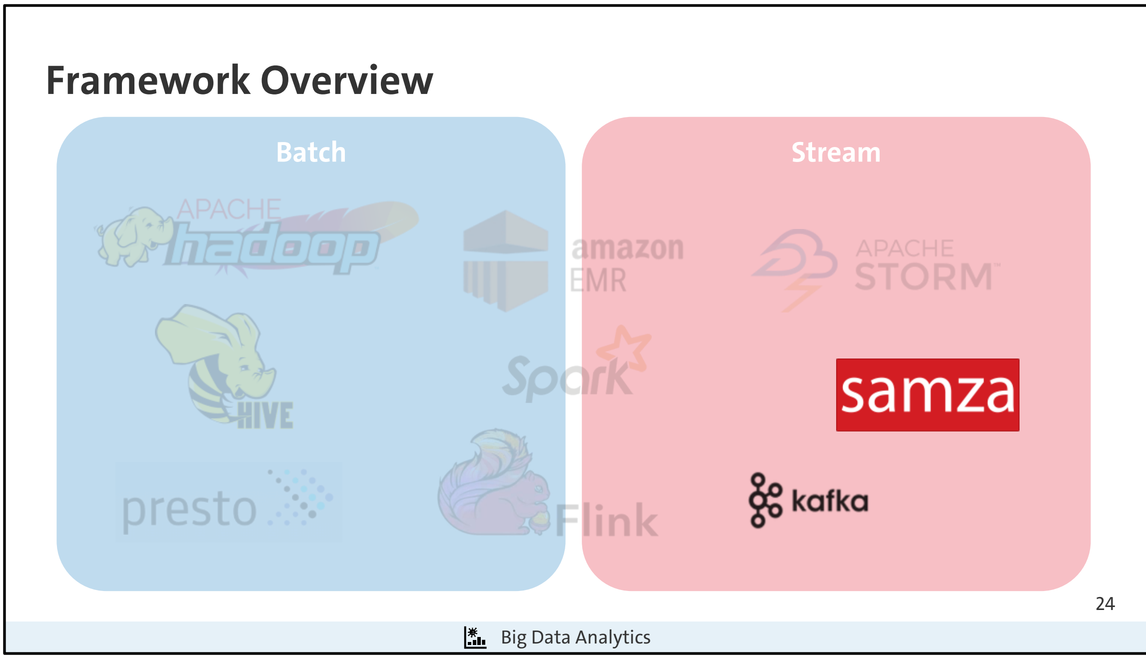
- Data warehouse on top of Map Reduce
- HiveQL SQL-like query language
- Supports different formats, e.g. Iceberg, RDD,...
- Goal: make Map Reduce more easily accessible

Apache Hive

```
CREATE TABLE page_view(viewTime INT, userid BIGINT,
                page_url STRING, referrer_url STRING,
                friends ARRAY<BIGINT>, properties MAP<STRING, STRING>,
                ip STRING COMMENT 'IP Address of the User')
COMMENT 'This is the page view table'
PARTITIONED BY(dt STRING, country STRING)
CLUSTERED BY(userid) SORTED BY(viewTime) INTO 32 BUCKETS
ROW FORMAT DELIMITED
        FIELDS TERMINATED BY '1'
        COLLECTION ITEMS TERMINATED BY '2'
        MAP KEYS TERMINATED BY '3'
STORED AS SEQUENCEFILE;
```


Example code from https://cwiki.apache.org/confluence/display/Hive/Tutorial

Comments can be added at column level and at table level



Kafka

- A storage format and a streaming platform (kappa architecture)
- Storage format is also used/can be processed by other systems, e.g. SAMZA, Flink,...

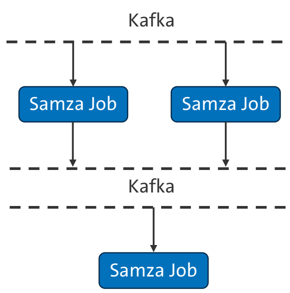
SAMZA

- Co-developed with Kafka
- Simple by design: only single-step jobs
- Local state
- Native stream processor: low latency
- Users: LinkedIn, Uber, Netflix, TripAdvisor, Optimizely, ...

<u>History</u>

- Developed at LinkedIn
- 2013: open-source (Apache Incubator)
- 2015: Apache top-level project

Samza

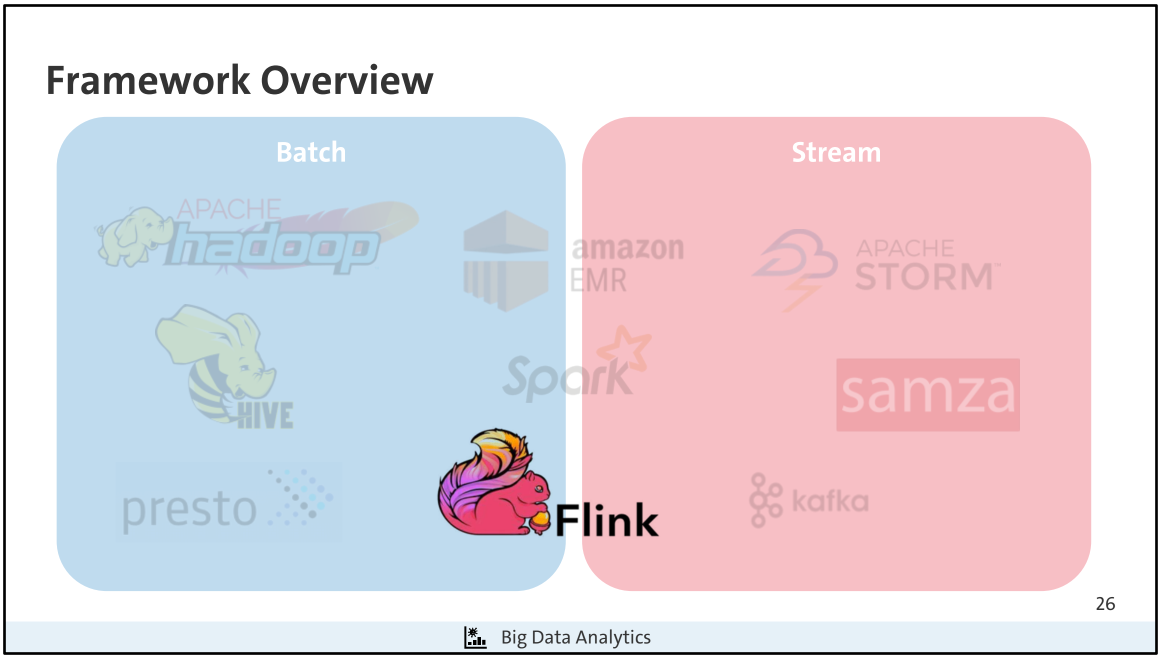


→ Local states enable simple recovery

Kafka Storage

- Not intended as permanent storage → Maximum time or amount of data can be defined before data is purged
- Stores events/messages in partitions which are segmented (usually 1GB)
- Compression can be enabled

- Job : processing step
  - → Robust
  - → But: often several jobs
- Task : Job instance (parallelism)
- Message : single data item
- Output persisted in Kafka
  - → Easy data sharing
  - → Buffering no back pressure
  - → But: Increased latency
- Ordering within partitions



Overview

- Native stream processor: Latency <100ms feasible
- Abstract API for stream and batch processing, stateful, exactly-once delivery
- Many libraries: Table and SQL, CEP, MachineLearning , Gelly…
- Users: Alibaba, Ericsson, Otto Group, ResearchGate, Zalando…

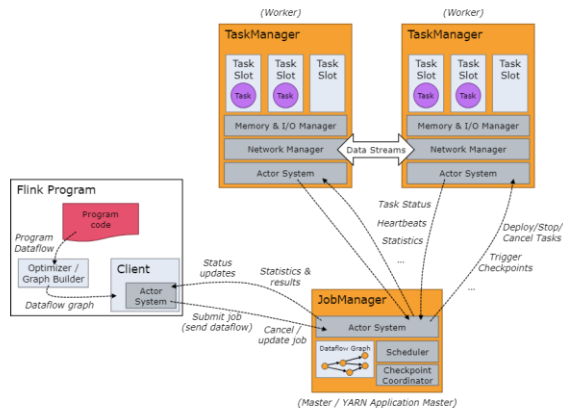
History

- 2010: Start as Stratosphere at TU Berlin, HU Berlin, andHPI Potsdam
- 2014: Apache Incubator, project renamed to Flink
- 2015: Apache top-level project

Flink

2 Execution Modes: Batch & Streaming

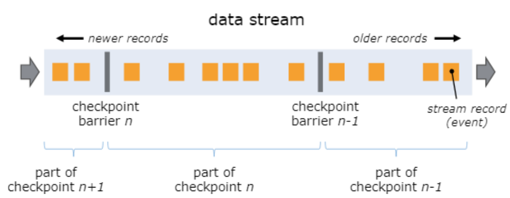
- Batch: Requires bounded jobs, i.e. all input coming form the source(s) is known before execution)
- Streaming: Can be used for bounded and unbounded jobs



https://nightlies.apache.org/flink/flink-docs-master/docs/concepts/flink-architecture/

Automatic backups of local state
→ Stored in RocksDB, Savepoints written to HDFS

Checkpointing in Flink (Stream processing)



Checkpoints are always between different records, they never overtake them
→ Intermediate operator emits a barrier *n* if it received the same barrier *n* from all its input streams
→ Final/sink operator acknowledges snapshot *n* to the checkpoint coordinator
→ A snapshot *n* is completed when all sink operators acknowledged *n*
→ No records from before snapshot n will be needed anymore

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Further Reading
Lightweight Asynchronous Snapshots for Distributed Dataflows, Carbone at al.
https://arxiv.org/abs/1506.08603

How to decide?

As always: It depends on the use case and its requirements
→ Tradeoffs have to be made

Latency vs. throughput
Latency vs. fault-tolerance
Simplicity vs. control
Historically grown system vs. completely new setup
Required features, e.g. state management, consistency guarantees, ordering,...
...

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG